Explicit State Space Verification

H A B I L I T A T I O N S S C H R I F T

zur Erlangung der Lehrbefähigung
im Fach Informatik

vorgelegt dem Rat der
Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von
Herr Dr. rer. nat. Karsten Schmidt
geboren am 3.3.1967 in Berlin

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jürgen Mlynek

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr. Peter Starke
2. Prof. Dr. Antti Valmari
3. Prof. Dr. Javier Esparza

eingereicht am: 4. Februar 2002
Tag des Fachgesprächs: 15. November 2002

Abstract

Verification is the task of determining whether a (model of a) system holds a given behavioral property. State space verification comprises a class of computer aided verification techniques where the property is verified through exhaustive exploration of the reachable states of the system. Brute force implementations of state space verification are intractable, due to the well known state explosion problem. Explicit state space verification techniques explore the state space one state at a time, and rely usually on data structures where the size of the data structure increases monotonously with an increasing number of explored states. They alleviate state explosion by constructing a reduced state space that, by a mathematically founded construction, behaves like the original system with respect to the specified properties. Thereby, decrease of the number of states in the reduced system is the core issue of a reduction technique thus reducing the amount of memory required.

An explicit state space verification technique comprises of

- a theory that establishes whether, and how, certain properties can be preserved through a construction of a reduced state space;
- a set of procedures to execute the actual construction efficiently.

In this thesis, we contribute to several existing explicit state space verification techniques in either of these two respects.

We extend the class of *stubborn set* methods (an instance of *partial order reduction*) by constructions that preserve previously unsupported classes of properties. Many existing constructions rely on the existence of "invisible" actions, i.e. actions whose effect does not immediately influence the verified property. We propose efficient constructions that can be applied without having such invisible actions, and prove that they preserve reachability properties as well as certain classes of more complex behavioral system properties. This way, so called "global" properties can now be approached with better stubborn set methods.

We pick up a graph automorphism based approach to *symmetry* reduction and propose a set of construction algorithms that make this approach feasible. In difference to established symmetry techniques that rely on special "symmetry creating" data types, a broader range of symmetries can be handled with our approach thus obtaining smaller reduced state spaces.

Coverability graph construction leads to a finite representation of an infinite state space of a Petri net by condensing diverging sequences of states to their limit. We prove rules to determine temporal logic properties of the original system from its coverability graph, far beyond the few properties known to be preserved so far.

We employ the Petri net concept of linear algebraic *invariants* for compressing states as well as for leaving states out of explicit storage. Compression uses place invariants for replacing states by smaller fingerprints that still uniquely identify a state (unlike many hash compression techniques). For reducing the number of explicitly stored states, we rely on the capability of Petri net transition invariants to characterize cycles in the state space. For termination of an exhaustive exploration of a finite state space, it is sufficient to cover all cycles with explicitly stored states. Both techniques are easy consequences of well known facts about invariants. As a novel contribution, we observe that both techniques can be applied without computing an explicit representation of (a generating set for) the respective invariants. This speeds up the constructions considerably and saves a significant amount of memory.

For all presented techniques, we illustrate their capabilities to reduce the complexity of state space reduction using a few academic benchmark examples. We address compatibility issues, i.e. the possibility to apply techniques in combination, or in connection with different strategies for exploring the re-

duced state space. We propose a scheme to distribute state space exploration on a cluster of workstations and discuss consequences for using this scheme for state space reduction. We collect observations concerning the impact of the choice of system description formalisms, and property specification languages, on the availability of explicit state space verification techniques.

Keywords:
Computer Aided Verification, State space analysis, Reduction techniques, Model Checking

Zusammenfassung

Gegenstand der Arbeit ist die Verifikation von verteilten diskreten Systemen in bezug auf Spezifikationen ihres Verhaltens. Diskrete Systeme bestehen aus einer abzählbaren Zustandsmenge und einer Zustandsübergangsrelation. Bei verteilten Systemen ist eine signifikante Zahl von Zustandsübergängen nur durch eine kleine Zahl von Komponenten eines strukturierten Zustandsraumes bedingt und ändert auch nur wenige Komponenten. Bei praktisch relevanten Systemen ist die Zustandszahl unbeherrschbar groß. Dieses Phänomen wird Zustandsraumexplosion genannt. Verteiltheit gilt als eine der wesentlichen Ursachen für Zustandsraumexplosion, weil nebenläufig mögliche lokale Zustandsübergänge abhängig von ihren exponentiell vielen Ausführungsreihenfolgen exponentiell viele verschiedene Zwischenzustände erzeugen können. Für Verifikationsaufgaben sind Systeme daher implizit gegeben durch eine Beschreibung von Anfangszuständen und (lokale) Regeln zur Generierung von Folgezuständen. Solche Systembeschreibungen folgen verschiedenen Paradigmen, z.B. dem variablenorientierten Paradigma (Zustände sind Werte von Variablen, die durch Zustandsübergänge gelesen und geschrieben werden) oder dem ressourcenorientierten Paradigma (Zustände sind Verteilungen von Ressourcen im System, die durch Zustandsübergänge konsumiert oder produziert werden). Die Verfügbarkeit von Verifikationstechniken oder spezifischen Implementationen hängt vom zugrundeliegenden Paradigma ab.

Als Sprache zur Formulierung von Spezifikationen des Verhaltens verwenden wir etablierte temporale Logiken und für die Praxis bedeutsame Fragmente solcher Logiken. Temporale Logik beschreibt Eigenschaften von Abfolgen von Zuständen, basierend auf elementaren, einzelne Zustände betreffenden Eigenschaften. Auf einer expliziten Systemdarstellung lassen sich temporallogische Eigenschaften effizient, d.h. mit einer linear von der Zustandszahl abhängigen Laufzeit, verifizieren. Eine solche Verifikation basiert auf einfachen Suchalgorithmen in dem durch das System definierten Zustandsgraph. Ein solcher Verifikationsansatz ist aber wegen der genannten Zustandsraumexplosion nicht durchführbar.

Im wesentlichen werden drei Lösungsansätze in Richtung durchführbarer Verifikationsalgorithmen verfolgt. Die strukturelle Verifikation versucht, Eigenschaften direkt aus spezifischen Mustern in der impliziten Systembeschreibung abzuleiten. Der derzeitige Stand der Technik gestattet solche Ableitungen nur für wenige und einfach strukturierte Verhaltensspezifikationen und erfordert auch dann in einigen Fällen recht aufwendige Berechnungen. Bei der symbolischen Zustandsraumanalyse wird der Zustandsraum erschöpfend durchmustert, allerdings unter Benutzung von Datenstrukturen, deren elementare Objekte ganze Mengen von Zuständen beschreiben, und deren elementare Operationen die Folgezustände für ganze solche Mengen aus der impliziten Systembeschreibung errechnen. Bei der expliziten Zustandsraumverifikation, dem Thema der vorliegenden Habilitationsschrift, wird eine explizite Repräsentation eines Zustandsraumes generiert, der wesentlich kleiner ist als der Zustandsraum des untersuchten Systems, in bezug auf die untersuchte Eigenschaft aber per Konstruktion äquivalent zum originalen System ist. Zur Konstruktion werden Informationen aus der impliziten Systembeschreibung herangezogen.

Eine Technologie zur expliziten Zustandsraumverifikation besteht also aus

- Einer mathematisch fundierten Theorie, die einer bestimmten Konstruktionsmethode bescheinigt, welche Eigenschaften durch sie bewahrt werden;
- effizienten Algorithmen zur Implementation eine solchen Konstruktion;

Die Arbeit enthält, für mehrere bekannte Verfahren, Beiträge zu jeweils mindestens einem der beiden Bestandteile einer expliziten Zustandsraumverifikationstechnik.

Die Methode der *sturen Mengen* verkleinert den explizit zu konstruierenden Zustandsraum dadurch, daß von den in einem Zustand möglichen Zustandsübergängen nur einige tatsächlich untersucht werden, so daß weit weniger Zwischenzustände durch verschiedene Abfolge nebenläufiger lokaler Zustandsübergänge entstehen. Die zu untersuchenden Übergänge werden abhängig von der zu verifizierenden Eigenschaft und Informationen aus der Systemstruktur so ausgewählt, daß zu jeder Klasse von für die Eigenschaft relevanten Systemabläufen wenigstens einer im reduzierten Zustandsraum repräsentiert ist. Die erste 1988 veröffentlichte Methode diente der Bewahrung von terminalen Zuständen sowie mindestens eines Pfades unendlicher

Länge. In der Folge wurde diese Technik auf viele andere Klassen von Eigenschaften erweitert, wobei vor allem die Fähigkeit, einen unendlichen Pfad zu bewahren, dahingehend verfeinert wurde, daß gezielt Pfade mit bestimmten Eigenschaften bewahrt werden konnten. Dabei spielte das Konzept unsichtbarer Zustandsübergänge eine tragende Rolle, wobei ein unsichtbarer Zustandsübergang die Eigenschaft hat, daß er keine für die Eigenschaft relevanten Zustandskomponenten ändert. Daher war die Anwendung der Methode sturer Mengen begrenzt auf lokale Systemeigenschaften, weil andereseits zu wenige unsichtbare Übergänge für eine substantielle Reduktion zur Verfügung stünden.

In der vorliegenden Arbeit setzen wir an der ersten Arbeit zur Methode sturer Mengen an und verfeinern die Fähigkeit, terminale Zustände zu bewahren, dahingehend, daß nun die Präsenz von Zuständen mit beliebigen in temporaler Logik formulierbaren Eigenschaften bewahrt werden. Die neue Methode basiert nicht auf der Existenz unsichtbarer Übergänge und kann in der Tat auch bei der Verifikation globaler Systemeigenschaften zu substantieller Reduktion führen. Das neue Konzept zur Konstruktion des reduzierten Zustandsraumes sind sogenannte UP-Mengen. Eine UP-Menge ist eine Menge von Übergängen, von denen mindestens einer in einem Systemablauf von einem Zustand, der die untersuchte Eigenschaft nicht erfüllt, zu einem Zustand, der die Eigenschaft erfüllt, vorkommen muß. Wir geben Algorithmen an, die kleine UP-Mengen für beliebige Zustände aus der impliziten Systembeschreibung und einer Repräsentation der untersuchten Eigenschaft in der temporalen Logik CTL berechnet. Wir zeigen, daß jede Konstruktion, die in einem Zustand alle Übergänge in einer schwach sturen Obermenge einer zu dem Zustand berechneten UP-Menge untersucht, alle Zustände erreicht, die die Eigenschaft besitzen. Dabei ist die Konstruktion schwach sturer Mengen die allen Methoden sturer Mengen gemeinsame Grundkonstruktion.

Symmetrische Reduktion verkleinert den zu untersuchenden Zustandsraum dadurch, daß zu jeder Klasse von in bezug auf Symmetrie äquivalenten Zuständen jeweils nur einer weiterverfolgt wird. Dadurch lassen sich alle gegenüber Symmetrie insensitive Eigenschaften bewahren (wobei man oft Insensitivität einer Eigenschaft durch die geeignete Wahl der Symmetrienmenge erreichen kann). Symmetrische Reduktion beinhaltet zwei Probleme, erstens das Aufinden der einem System innewohnenden Symmetrie, und zweitens, zu einem gegebenen Zustand, das Auffinden zu ihm äquivalenter Zustände in der Menge bereits untersuchter Zustände. Die meisten vorhande-

nen Implementationen leiten Symmetrien aus speziellen Datenstrukturen ab, in denen wegen der eingeschränkten Operationen die verschiedenen Elemente des Typs austauschbar sind. Das Auffinden äquivalenter Zustände wird durch eine Transformation neu berechnter Zustände in einen äquivalenten kanonischen Repräsentanten realisert. Alternativ zu diesem Ansatz wurde zur Beschreibung von Symmetrien die Verwendung von Graphautomorphismen auf netzartigen impliziten Systembeschreibungsformen vorgeschlagen. Es zeigt sich, daß per Umwandlung von Datenabhängigkeiten in Graphrepräsentationen, jede Datentypsymmetrie auch einen Graphautomorphismus bildet, andererseits aber durch Graphautomorphismen Symmetrien beschreibbar sind, die sich in Datentypbetrachtungen nicht wiederfinden lassen. Diese zusätzlichen Symmetrien erlauben eine stärkere Reduktion des Zustandsraumes. Zur Graphautomorphismentechnik fehlten bislang leistungsfähige Algorithmen zur Umsetzung dieser Technologie.

Wir setzen an der auf Graphautomorphismen basierenden Methode an und unterlegen alle Teilprobleme mit leistungsfähigen Algorithmen. Die Berechnung der Automorphismen beschränken wir auf ein Erzeugendensystem, das polynomiell viele Elemente, gemessen an der Größe der impliziten Systembeschreibung, hat. Die Berechnung selbst ist schlimmstenfalls exponentiell, was nicht verwundert, weil das Problem mit einem Entscheidungsproblem eng korreliert, von dem bekannt ist, daß es in der Klasse NP, aber unbekannt, ob es NP-vollständig oder in P liegt. Diese Eigenschaft hat dem Problem eingehende Untersuchung zuteil werden lassen, wegen der nach wie vor offenen „P $\neq$ NP?“-Frage. Trotzdem ist kein polynomieller Algorithmus bekannt. Umso erfreulicher ist es, daß unser Berechnungsalgorithmus sich auf realistischen Beispielen bisher durchweg polynomiell verhielt, und lediglich bei eigens konstruierten Systemen ins Exponentielle ausriß. Für die Lösung des Problems, äquivalente bereits bekannte Zustände aufzuspüren, schlagen wir mehrere Techniken vor und beschreiben ihre Leistungsfähigkeit abhängig von der Struktur der innewohnenden Symmetrie. Für dünne Symmetriegruppen (wenige symmetrische Transformationen) eignet sich eine Technik, bei der die Symmetrien der Reihe nach aus dem Erzeugendensystem generiert werden, und das symmetrische Bild des neuen Zustandes mit der Menge der bekannten Zustände verglichen wird. Dabei können wir, abhängig vom Ausgang einer solchen Überprüfung, die Generierung von Symmetrien unterdrücken, von denen aus vorhandenen Informationen klar ist, daß sie keinesfalls zum Erfolg führen. Dadurch kann eine erhebliche Effizienzsteigerung erzielt werden. Bei einer zweiten Technik iterieren wir die bekannten

Zustände, genauer gesagt, diejenigen Zustände, die für eine die Symmetrie respektierende Hashfunktion denselben Wert liefert wie der neue Zustand, ob es eine Symmetrie gibt, die beide Zustände ineinander überführt. Das verbleibende Problem kann durch eine Adaption des Symmetrieberechnungsverfahrens gelöst werden. Eine vorherige Berechnung des Erzeugendensystems kann entfallen. Die dritte vorgeschlagene Technik benutzt das Erzeugendensystem, um den neuen Zustand approximativ in einen kanonischen äquivalenten Zustand zu überführen. Diese Technik ist von allen beschriebenen Methoden die effizienteste, liefert aber größere Zustandsräume als die beiden anderen Techniken. Wir studieren die Vor- und Nachteile aller Techniken anhand mehrerer Beispielsysteme.

Die dritte in der Arbeit behandelte Technik ist die Methode der *Überdeckbarkeitsgraphen*. Sie ist spezifisch für die ressourcenbasierte Systembeschreibungsform der Petrinetze. Sie diente ursprünglich zur Aufspürung von Stellen im System, an denen sich unbeschränkt viele Ressourcen ansammeln können. Formal ist ein Überdeckbarkeitsgraph eine endliche Abstraktion eines Systems mit bis zu unendlich vielen Zuständen. Von nur wenigen Eigenschaften war bekannt, daß sie sich aus dem Überdeckbarkeitsgraphen ableiten lassen.

Wir formulieren Regeln zur Auswertung von Überdeckbarkeitsgraphen, mit deren Hilfe es möglich ist, eine Vielzahl von in temporaler Logik formulierten Eigenschaften aus dem Überdeckbarkeitsgraph abzuleiten. Diese Reglen sind inhärent unvollständig, da bekannt ist, daß für viele Eigenschaften es Paare von Systemen gibt, die isomorphe Überdeckbarkeitsgraphen liefern, sich aber in bezug auf die Eigenschaft verschieden verhalten. Für universelle Eigenschaften des CTL-Fragments ACTL erhalten wir Bewahrungsresultate durch das Ausweisen einer Simulationsrelation zwischen dem originalen System und seinem Überdeckbarkeitsgraph. Für existentielle Eigenschaften basieren unsere Resultate auf einer Abschwächung der Erfüllbarkeitsrelation über Zuständen des Überdeckbarkeitsgraphen. Einem Zustand des Überdeckbarkeitsgraphen entsprechen divergierende Folgen von Zuständen des Originalgraphen. Normalerweise schreibt man einem Zustand des Überdeckbarkeitsgraphen dann eine Eigenschaft zu, wenn alle Folgenglieder im Originalsystem die Eigenschaft besitzen. Wir arbeiten dagegen mit einem Begriff, wo Gültigkeit der Eigenschaft nur für fast alle Folgenglieder gefordert wird.

Eine letzte Gruppe von Techniken ist bisher in der Zustandsraumverifika-

tion nicht eingestzt worden, aber aus der strukturellen Verifikation für Petrinetze bekannt. Zu einem Petrinetz kann eine ganzzahlige Inzidenzmatrix C gebildet werden, mit deren Hilfe ein linear-algebraischer Zusammenhang zwischen voneinander errichbaren Zuständen hergestellt werden kann. Stellen- und Transitionsinvarianten sind Lösungen der durch C^T bzw. C definierten homogenen Gleichungssysteme. Dabei dienen Stelleninvarianten gewöhnlich einer Abschätzung der Menge der erreichbaren Zustände nach oben, mit daraus resultierenden Möglichkeiten der Ableitung von Eigenschaften, während Transitionsinvarianten Zyklen im Zustandsraum charakterisieren.

Wir verwenden Stelleninvarianten zur Kompression von einzelnen Zuständen. Durch Stelleninvarianten lassen sich einige Komponenten in einen funktionalen Zusammenhang zu den verbleibenden Komponenten stellen. Dadurch ist auch nach dem Streichen der funktional abhängigen Stellen der Zustand noch eindeutig determiniert. Wir zeigen, daß bei der Konstruktion des Zustandsraumes ein durch die verbleibenden Stellen gebildeter „Fingerabdruck“ ausreicht. Transitionsinvarianten verwenden wir dazu, eine Menge von Zuständen so auszuzeichnen, daß jeder Zyklus im Zustandsraum mindestens einen ausgezeichneten Zustand enthält. Darufhin speichern wir noch noch ausgezeichnete Zustände permanent, sparen also Speicherplatz. Für nicht ausgezeichnete Zustände kann es passieren, daß sie mehrmals aufgesucht werden (auf verschiedene Weise aus Vorgängerzuständen entstehen). Weil sie nicht gespeichert sind, werden auch wiederholt ihre Nachfolgezustände untersucht. Da in jedem Kreis mindestens ein ausgezeichneter, also permanent zu speichernder Zustand enthalten ist, entstehen durch diese wiederholte Berechnung keine Probleme in bezug auf Terminierung des Verfahrens, wohl aber erhebliche Laufzeiteinbußen. Wir schlagen Methoden zur Begrenzung der Laufzeiteinbußen um den Preis weiterer zu speichernder Zustände vor.

Für alle untersuchten Methoden studieren wir die Abhängigkeit der Anwendbarkeit und Effizienz der Methode von dem der gegebenen impliziten Systembeschreibung zugrundeliegenden Paradigma. Wir untersuchen ebenfalls die Kompatibilität der Verfahren mit verschiedenen Strategien zur Generierung des Zustandsraumes (Tiefe zuerst, Breite zuerst, verteilt) und Möglichkeiten der gemeinsamen Anwendung verschiedener Techniken.

Schlagwörter:
Computergestützte Verifikation, Zustandsraumanalyse, Reduktionstechniken, Modelchecking

Contents

I Prerequisites **3**

1 Systems **5**
- 1.1 Structure and behavior . . . 5
- 1.2 System properties . . . 14
- 1.3 Fairness . . . 19
- 1.4 Relations between systems . . . 20

2 Performance evaluation of verification algorithms **23**
- 2.1 Worst case and average case analysis . . . 23
- 2.2 Industrial case studies . . . 25
- 2.3 Academic benchmarks . . . 27
- 2.4 Running examples . . . 30

II State space exploration **32**

3 Search strategies **34**
- 3.1 Depth first search and detection of strongly connected components . . . 35
- 3.2 Implementation issues . . . 37
- 3.3 Breadth first search . . . 41
- 3.4 Distributed search . . . 43

4 Explicit state space verification **55**
- 4.1 Reachability properties . . . 55
- 4.2 Home properties . . . 57
- 4.3 CTL model checking . . . 61
- 4.4 LTL model checking . . . 70

4.5 Liveness properties 71

III Reduction techniques 76

5 Stubborn sets 78
5.1 Theory 81
5.2 Algorithms 97
5.3 Performance 101
5.4 Compatibility 106
5.5 Discussion 106

6 Symmetries 109
6.1 Theory 113
6.2 Algorithms 118
6.3 Performance 132
6.4 Compatibility 137
6.5 Discussion 140

7 Coverability analysis 142
7.1 Theory 143
7.2 Algorithms 153
7.3 Performance 154
7.4 Compatibility 155
7.5 Discussion 156

8 Linear algebraic reduction 158
8.1 Theory 159
8.2 Algorithms 162
8.3 Performance 165
8.4 Compatibility 167
8.5 Discussion 169

IV Conclusions 170

9 Comparison with other verification tools 171
9.1 SMV 171
9.2 Spin 172

9.3 Murϕ . . . 173

10 Explicit state space verification **175**
10.1 Reachability properties . . . 175
10.2 Home properties . . . 176
10.3 LTL model checking . . . 177
10.4 ACTL model checking . . . 178
10.5 CTL model checking . . . 178
10.6 Liveness properties . . . 179

11 Discussion **180**
11.1 Combined use of explicit state space methods . . . 180
11.2 Resource oriented versus variable oriented system description . 182
11.3 Explicit versus symbolic verification . . . 183
11.4 State space versus structural verification . . . 184
11.5 Open problems . . . 184

List of Figures

1.1 Guarded commands and labeled transition system 7
1.2 System in primed variable style 8
1.3 System in mixed style 9
1.4 A Petri net 9
1.5 A computation tree 17

2.1 The dining philosophers 29
2.2 The readers and writers example 30

3.1 Depth first search with scc detection 36
3.2 Cycle detection in depth first search 38
3.3 A decision tree 41
3.4 A logically distributed decision tree 45
3.5 Local decision trees 47
3.6 Reaction to a search request 50
3.7 Effect of a state delegation 52

4.1 Depth first reachability verification 56
4.2 Depth first home property verification 62
4.3 Two situations in searchAU 66
4.4 Depth first evaluation of universal until 67
4.5 Depth first evaluation of existential until 68
4.6 Verification of CTL formulas 69
4.7 Search for fair strongly connected sets 75

5.1 State explosion through concurrency 79
5.2 Partial order reduction 80
5.3 Computation of strong stubborn sets 97

6.1 A graph with 48 automorphisms 111

6.2 Unfolding of a HL Petri net to a LL Petri net 112
6.3 Running example for illustrating REFINE and DEFINE . . . 119
6.4 A refine step 121
6.5 Computation of automorphisms 124
6.6 Integrating symmetries by iterating symmetries 129
6.7 Integrating symmetries by iterating states 130
6.8 Integrating symmetries by canonicalizing states using generators 131
6.9 The ECHO algorithm 134

7.1 An infinite state Petri net 144
7.2 Karp/Miller coverability graph 144
7.3 Finkel coverability graph 145

8.1 Depth first search using transition invariant based reduction . 164
8.2 Parikh vector guided state space exploration 166

List of Tables

5.1 UP and DOWN sets . . . 90
5.2 Reduction for reachability of unsatisfiable predicates . . . 102
5.3 Reduction for reachability of satisfiable predicates . . . 102
5.4 Home properties for PHi system . . . 104
5.5 Home properties for DAi system . . . 104

6.1 Automorphisms of the unit cube . . . 115
6.2 Symmetry preprocessing I . . . 133
6.3 Symmetry preprocessing II . . . 135
6.4 Reduced graph generation by iterating symmetries: PH n . . . 135
6.5 Reduced graph generation by iterating symmetries: DA n . . . 136
6.6 Reduced graph generation by iterating symmetries: SIMPLE d/n . . . 136
6.7 Reduced graph generation by iterating symmetries: ECHO d/n 136
6.8 Reduced graph generation by iterating states: PH n . . . 137
6.9 Reduced graph generation by iterating states: DA n . . . 137
6.10 Reduced graph generation by iterating states: SIMPLE d/n . 137
6.11 Reduced graph generation by iterating states: ECHO d/n . . . 138
6.12 Reduced graph generation by canonicalizing states: PH n . . . 138
6.13 Reduced graph generation by canonicalizing states: DA n . . . 138
6.14 Reduced graph generation by canonicalizing states: SIMPLE d/n . . . 139
6.15 Reduced graph generation by canonicalizing states: ECHO d/n 139

7.1 Limes-satisfiability of atomic propositions . . . 148
7.2 Coverability graph generation . . . 155

8.1 Run time for place invariant compression . . . 165

8.2 Run time for transition invariant based reduction—the PHi example . . . 167
8.3 Run time for transition invariant based reduction—the DAi example . . . 167
8.4 Performance of the transition invariant method in connection with stubborn sets. . . . 167

9.1 Performance of SMV . . . 172
9.2 Performance of Spin on a local reachability property . . . 173
9.3 Performance of Murϕ on a system with dense symmetry. . . . 173

11.1 Verification using combined reduction . . . 181

Preface

This thesis digests most of my research efforts since 1997. By that time, my research interests shifted from verification algorithms for a complex system description language (algebraic Petri nets) to the much simpler yet challenging formalism of place/transition Petri nets. Unlike with algebraic nets, my new ideas were much easier to implement, and implementation was inevitable for creating tables of "experimental results"that are compulsory elements of a publication in the field. First, the Petri net tool INA used to provide the natural environment for my algorithmical exercises. INA hosts a broad collection of verification techniques, including structural and state space techniques, and supports various Petri net formalisms.

As time went by, I recognized that the broad range of methods and formalisms available in INA caused several tradeoffs in the design of core data structures and procedures and limited the efficiency of particular verification algorithms. Aiming at competitive data in those "experimental results"tables, I started to implement the data structures and procedures I needed myself in an ad hoc fashion, independently of INA. Studying more and more different explicit state space techniques, I was surprised what large amount of code could be reused for new algorithms. From an implementation point of view, all explicit state space verification methods are singular modifications to a general search algorithm. It turned out that different techniques modified disjoint parts of the search algorithm, so it became apparent that they can be applied jointly. By combining all available methods, I was able to successfully verify systems that are orders of magnitude larger than those INA could handle.

Eventually, in the end of 1998, I decided to call my collection of ad hoc implementations *tool*, and I named it LoLA—a low level analyzer. From that time on, I continued extending the list of features in LoLA, with emphasize on availability of verification techniques for a broad list of properties much smaller than the temporal logics CTL and LTL. By that time, advanced technology for model checking temporal logic properties was already available. I concluded that the largest room left by these technology was dedicated, and thus more efficient solutaions to smaller problems. The impact of the size of a problem, be it the expressive power of a system description formalism or be it the class of supported properties, on the availability or efficiency of a verification method is therefore a recurring theme in this thesis.

Much of the technical material included here has already been published

in journals [Sch99a, Sch00a, Sch00e, Sch00d], or presented at international conferences [Sch99b, Sch00b, Sch00c]. However, this thesis is not meant to be just a collection of papers. In fact, virtually all of the text is rewritten. One reason for rewriting is that time lead me to additional insights, or a new approach to presenting a method. Another reason is that the more monographical shape of a thesis of this size required me to sustain from addressing every single detail of the methods that I am going to consider. Thus, I replaced some merely technical considerations with references to original papers. Discussion sections address more explicitly the recurring themes of this thesis. Basic notations and definitions relate to the whole thesis. Most experimental results have been gathered on one and the same machine.

Daniel Kröning implemented most of the procedures in LoLA that concern distributed search. The data structure described in Sec 3.3.4 for distributed search is the result of an intense discussion process with Daniel and should therefore be considered joint work of Daniel Kröning and the author of this thesis.

I would like to thank all my colleagues at Helsinki University of Technology, Technical University Dresden, Humboldt-Universität zu Berlin, and Carnegie Mellon University Pittsburgh for their support, inspiration, and the creative working atmosphere that I experienced during the past 6 years.

Part I

Prerequisites

In the chapters of this part, we carve out the area this thesis is concerned with. For this purpose, we introduce some major categories that distinguish the various approaches to system verification. We motivate our decisions about leaving some of these approaches out of our scope. Then, we set up the stage for the technical part of this thesis by defining central concepts. We spend some time at the end of this part to motivate the particular approach we shall use for illustrating the performance of the various methods we are studying subsequently.

Chapter 1

Systems

1.1 Structure and behavior

Without further clarification, the term *system* is highly ambiguous. For our purposes, systems are abstract views on real or imagined dynamic processes in nature, technology, or society. These processes can usually be observed (at least hypothetically) in terms of quantities that change over time. The major tool of abstraction is the concept of *state*. A state comprises all quantities relevant to the further evolution of a system. If these entities suffice to determine the future of a system uniquely, we call this system *deterministic*, otherwise *nondeterministic*. Nondeterminism is an inevitable consequence of abstraction, independently of our believes about existence of nondeterminism in the real world.

Systems can be distinguished by the nature of the quantities comprising their states. If all quantities range over dense domains, such system is called *continuous*. If all entities range over countable domains, the system is called *discrete*. If states comprise both discrete and continuous quantities, we call such a system *hybrid*. Continuous systems are widely and successfully used for studying processes in nature. Hybrid systems have recently raised attention of control theorists, since, in the context of embedded systems, the behavior of discrete controllers in a continuous environment is a major concern. In the digital realm of computer science, various processes can be modeled by means of discrete systems. States of a *real-time* system contain time as a particular quantity, and otherwise solely discrete quantities. Real-time systems that assume a discrete time scale fall into the category of

discrete systems while real-time systems that assume a dense time scale are hybrid.

Continuous and hybrid systems typically have uncountable state spaces (i.e. sets of states that they pass during their evolution). This makes them intractable for explicit state space verification without further abstraction since explicit state space verification is essentially about enumerating single states Thus, we confine all our forthcoming considerations to discrete systems.

In a reasonable discrete system, state changes occur only at singular points of time. These time points can then be referred to as *events*. Once having introduced the notion of events, the evolution of a system can be regarded as a finite or infinite sequence of events, and it is possible to abstract from an explicit notion of time. When reasoning about systems, it is convenient to tie together events supposed to be related to a common cause to an *action*. Using the concepts of state, event, and action, discrete systems can be formalized as *labeled transition systems*, a class of automata.

Definition 1 (Labeled transition system) *A* labeled transition system $[S, E, A]$ *consists of a countable set* S *of* states*, a countable set* A *of* actions*, and a set* E *(*$E \subseteq S \times A \times S$*) of* events[1]*.*

Some set I of states of a labeled transition system may be qualified as *initial* states, extending the notation of labeled transition systems to $[S, E, A, I]$. The relation $[s, a, s'] \in E$ can be written as $s \xrightarrow{a} s'$. We write $[s, s'] \in E$, or $s \rightarrow s'$ for: there exists an action a such that $[s, a, s'] \in E$. The arrow notion for events can be extended to finite sequences of actions by defining $s \xrightarrow{\varepsilon} s$ for arbitrary states s and the empty sequence ε, and defining $s \xrightarrow{wa} s'$ iff there is a state s'' such that $s \xrightarrow{w} s''$ and $s'' \xrightarrow{a} s'$. We say that s' is *reachable* from s ($s \xrightarrow{*} s'$) iff there is a sequence w of actions holding $s \xrightarrow{w} s'$. For a labeled transition system $[S, E, A, I]$ with initial states, one usually requires that S consists only of states that are reachable from some state in I.

An action a is *enabled* at a state s iff there is a state s' such that $s \xrightarrow{a} s'$. Action a is *deterministic* iff for every state s there is at most one s' holding $s \xrightarrow{a} s'$. It is *invertible* iff for every state s' there is at most one state s holding $s \xrightarrow{a} s'$.

[1] In the literature, the term *transition* is usually used instead. We decided to stick to *event* in order not to overload forthcoming definitions of *transition*.

Labeled transition systems are a reasonably simple formal approach to discrete systems. It is fairly easy to use them for the formal definition of system properties (see next section), or to devise decision procedures for such properties in the case of *finite state systems* (systems with finite S). They link the concept of discrete systems to the powerful theory of automata. Labeled transition systems have, however, a prohibitively large number of states in most cases of practical relevance, and even for a large number of toy examples. This phenomenon, known as the *state explosion problem*, calls for other, more concise formalisms to be used in the actual design of systems. Consequently, it is these formalisms that systems are given in as input to a verification algorithm. Thus, we account some system description formalisms.

Various formalisms set up upon a set of variables each ranging on a data domain. A state is an assignment of feasible values to these variables. In the guarded commands approach [Dij76], actions are formalized one by one by specifying their enabling condition (a boolean valued expression over the state variables), and their effect (a concurrent assignment of new values to the state variables; the new values are expressions over the state variables themselves).

```
var x,y : natural;

g1:  even(x) --> x := x DIV 2;
g2:  x < y   --> x := x+1; y := y-1;
g3:  y > 0   --> y := y-1;
```

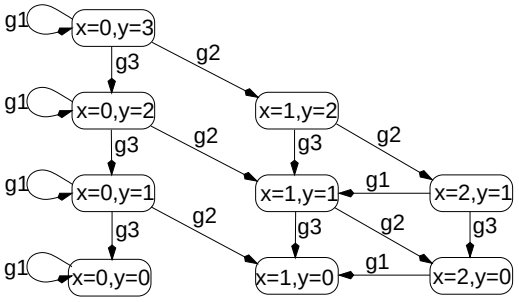

Figure 1.1: Guarded command program and labeled transition system, assuming $x = 0$ and $y = 3$ as initial state.

Every guarded command program, consisting of a list of variable declaration and a set of guarded commands, defines a labeled transition system. In this system, all feasible assignments to the declared state variables serve as states. Every guarded command g defines a set of events $[s, g, s']$ such that the enabling condition of g becomes true in s, and in s' variables are replaced by the values of the assigned expression (evaluated using values from s). Variables not mentioned in a guarded command are assumed not to change values.

Traditional high level programs without recursion can be rephrased as guarded command programs. It is sufficient to introduce a new state variable and assign values to it that correspond to control flow locations of the original program.

Actions defined by guarded commands are deterministic (at least, it is not common to have nondeterministic operations occurring within guarded command expressions). Usually, they are not invertible, though. For instance, two states differing in their value of a variable x can share the same successor via an action that contains a constant assignment such as `x := 0`.

Several verifiers, among them SMV [McM93], Spin, [Hol91], and Murϕ [DDHC92], support input of system descriptions in guarded command style. They differ in details of the execution semantics (whether one command is executed at a time, as described above, or whether all enabled guarded commands are executed simultaneously).

In another paradigm of system description, a copy of each variable is introduced (usually as the primed version of the original variable). The set of events in the system is then formalized as a boolean valued expression, called *transition relation*, that ranges over both sets of variables. An event from a state s to a state s' is associated to having the assignments of s to the original variables, and s' to the primed variables let the expression evaluate to true. Among others, SMV, and Lamport's TLA [Lam94] support this approach. In the transition relation approach, there is no canonical notion of action. Actions can either be left out completely, specified separately, or be introduced by having a separate transition relation for each action. In general, transition relations yield nondeterministic actions.

$$(x = 2{\cdot}x' \wedge y' = y) \vee (x < y \wedge x' = x{+}1 \wedge y = y'{+}1) \vee (y > 0 \wedge y = y'{+}1 \wedge x' = x)$$

Figure 1.2: The same system as above, written in primed variable style

Variables can also be used to get only partially away from an explicit, i.e. graphical notion of labeled transition systems. A skeleton labeled transition system graph having a vertex for each state and an edge for each event, can be annotated by variables (not attached to a particular graphical object) while expressions are attached to graphical objects. A state of the actual system consists of a state of the skeleton, plus an assignment to the variables. An

event corresponds to replacing a vertex by a successor vertex, and rewriting the variables according to assignments annotated at the taken edge. Boolean expressions annotated at the edge (*transition guards*) , or at the successor vertex (*state invariant*) can prevent an edge from being taken in a state where the variable values render such expressions false. The real-time verification tool UPPAAL [LPY97] has an input language using this paradigm. Actions in this approach are deterministic if the skeleton is deterministic and the assignments do not introduce nondeterminism.

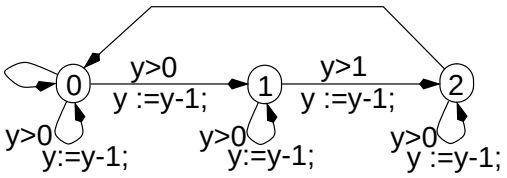

Figure 1.3: Same system in mixed explicit and variable based notation; y assumes 3 as initial value

Instead of variables, one can use *locations*, or *places*, as carriers of state. A state is then a distribution of *resources* over the locations. For an action, one would specify which resources it consumes or produces at each locations. In this view, an action would be disabled at least in states where not all resources to be consumed are present in the respective locations. Petri nets, the most popular formalism of this class, have absence of resources as their only way to disable actions. Actions in resource oriented descriptions are usually deterministic and, at least in the Petri net case, as well invertible.

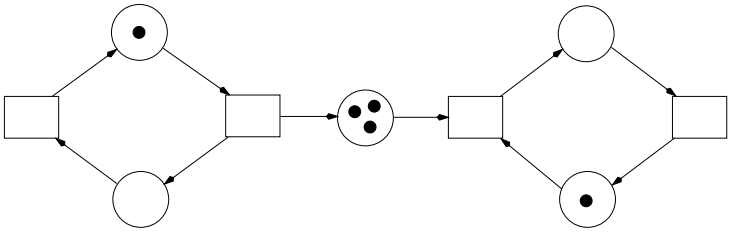

Figure 1.4: Not the same system, depicted as Petri net; places are circles, actions are boxes, resources are black dots, and consumption/production is coded by edges

Systems can be composed to larger systems. In *sequential composition*, only one component is active, and control is passed to the second component

when the first component terminates. The second component may access the final values of the variables in the first component. In *parallel composition*, both components are active at the same time. The composed system can be executed *synchronously* (an event in the composed system corresponds to the simultaneous occurrence of one event per component) or *asynchronously* (an event in the composed system correspond to only one event of one of its components). Interaction of parallel components is more complex than interaction of sequential components. Components may exchange data via variables that both can access (*shared variables*)), or via passing messages to each other. In the message passing case, there are dedicated *send* and *receive* actions and message buffers between the components. Upon *send*, the sending component would add a data record to the buffer. The enabling condition of a send action includes the test whether a buffer cell is available. A *receive* action retrieves a record from the buffer and is disabled when the buffer is empty. The system behavior depends on the specified buffer sizes. An unlimited buffer would never disable a send action. Buffers can be skipped altogether if messages are passed by simultaneous occurrence of *send* and *receive*. This kind of synchronizing actions of different component is, besides semaphores for assuring mutual exclusion, one of the major ways to interfere with another component's control flow.

We conclude this (incomplete) enumeration of system description techniques with just mentioning the possibility of imposing refinement techniques (replacing elementary syntactical units by whole components) and hierarchies (having dedicated syntactical units for representing components and their interfaces inside other components) for further structuring system descriptions.

Faced with a Babylonian confusion of system description formalisms, it is now our task to pick one or more languages as the underlying formalism(s) for our further discourse. Just sticking to the greatest common denominator—labeled transition systems—is not an option since bridging the gap between a system description and the labeled transition system it defines is what this thesis is about. Giving all approaches a fully formal treatment would, on the other hand, consume an inappropriately large amount of space. Studying one formalism certainly covers other formalisms, at least to the degree descriptions can be transferred between formalisms. Possibilities of translating descriptions are far reaching but incomplete. This can be easily recognized by comparing the decidable reachability problem for Petri nets with the undecidable reachability problem for guarded command languages having prim-

itives as usual programming languages (the halting problem is a reachability problem). The finite state versions of the formalisms (in the sense of finite variable domains, or buffers and locations having finite capacity) mentioned earlier have, however, equivalent expressive power. This is basically due to the fact that all formalisms allow for a one-to-one representation of arbitrary finite labeled transition systems.

The capability of expressing arbitrary labeled transition systems extends only to states and events, but not necessarily to actions. Obviously, a formalism having invertible actions only, like Petri nets, is incapable of representing any labeled transition system that features non-invertible actions. This incompatibility can be fixed by letting a set of actions of the host formalism model a single action of the system to be modeled. Technically, this can be achieved by (noninjectively) labeling the actions of the host formalism and treating the labels as action rather than the elementary syntactical units. For our goals, this view is not appropriate, though. We view actions primarily as a tool for generating events out of a system description. These events are generated out of the elementary syntactical units of the underlying formalism, no matter what labeling is imposed on top of them.

Revisiting translations between formalisms in the light of a strict preservation of actions (where one action of the original description corresponds to a unique action of the target description), we can now call a formalism A more flexible than a formalism B if every description in B can be effectively transformed to an action-preserving equivalent description in A. Now, suppose there is a state space verification technique for the more flexible formalism that is based on exploring new states by computing events out of the system's description of actions. Such a technique could be easily adapted to work for every less flexible formalism, too (if everything else fails, simply put the translation in front of the original technique). In contrast, for the less flexible formalism there may be techniques that take advantage of specific properties of actions in that formalism and are not applicable in the more flexible one.

That is, if we would base our considerations on a most flexible system description formalism (one with a strong expressive power), we could study only a small number of formalisms. We find it therefore wiser to pick a formalism that is rather restrictive in its modeling power yet popular enough to be safe of totally irrelevant techniques. This way, a larger number of verification techniques can be treated in a uniform formal framework. We shall, however, discuss to which degree each technique relies on the specific

features of the formalism of our choice, and the implications for applying the same technique to more general system description languages. We find this approach the best way to cover both a large number of system description languages and a large number of verification techniques yet keeping the formal approaches readable.

Among the formalisms enumerated earlier, we find that Petri nets are the perfect candidate for our purposes. Petri net actions are not only deterministic, but also invertible (in contrast to most variable based languages). Petri nets can be easily translated to the guarded command framework and other variable oriented formalisms such that actions are preserved. Other advantages of using Petri net include

- simplicity
- linearity
- monotonicity

where at least the last two properties can hardly be found in other formalisms. Simplicity helps us to keep proofs readable. Linearity means that the *difference* between a state and its successor state via an action does not depend on the involved states. This phenomenon enables linear algebraic techniques, see Ch. 8. Monotonicity means that the enabling condition of actions is compatible with a natural partial order on states in the sense that an action enabled in some state is enabled in all larger states, too. The coverability graph technique (Ch. 7) relies on that property.

Note that the argumentation above is limited to picking a language for presenting various verification techniques in a uniform formal framework. For choosing a formalism as a tool's input language, or for actually conducting a case study, many other issues would be relevant.

Definition 2 (Petri net) *A Petri net* $N = [P, T, F, W, m_0]$ *consists of a finite set* P *of* places*, a disjoint finite set* T *of* transitions*, a set* $F \subseteq (T \times P) \cup (P \times T)$ *of* arcs *connecting places with transitions and vice versa, a mapping* $W : F \longrightarrow \mathbf{N}^+$ *that assigns an* arc weight *to each arc, and an* initial marking m_0*. A* marking $m : P \longrightarrow \mathbf{N}$ *assigns a number of* tokens *to each place.*

Places and transitions are collectively called *nodes*. For a node $x \in P \cup T$, ${}^\bullet x$ denotes its *pre-set* ${}^\bullet x = \{y \mid [y, x] \in F\}$ and $x^\bullet$ denotes its *post-set* $x^\bullet = \{y \mid [x, y] \in F\}$. For $[x, y] \notin F$, define $W([x, y]) = 0$.

Places are the carrier of Petri net states, transitions represent actions, weighted arcs define the effect of an action as well as its enabling condition, and markings are a synonym for states.

Definition 3 (Enabledness, state space) *Let $N = [P, T, F, W, m_0]$ be a Petri net. Transition $t \in T$ is enabled at marking m iff for all $p \in {}^\bullet t$, $m(p) \geq W([p,t])$. Marking m' is* directly *reachable from marking m via transition t ($m \xrightarrow{t} m'$) iff t is enabled at m and for all $p \in P$, $m'(p) = m(p) - W([p,t]) + W([t,p])$. We extend reachability inductively to transition sequences: let $m \xrightarrow{\varepsilon} m$ for the empty sequence ε, and let $m \xrightarrow{wt} m'$ for a sequence w and a transition t iff there is a marking m_1 such that $m \xrightarrow{w} m_1$ and $m_1 \xrightarrow{t} m'$. Marking m' is* reachable *from marking m ($m \xrightarrow{*} m'$ iff there is a transition sequence w such that $m \xrightarrow{w} m'$.*

The introduced arrow notations for reachability are compatible with the corresponding notions for the labeled transition system that is canonically associated to a Petri net— it's *state space*.

Definition 4 (Petri net state space) *Let $N = [P, T, F, W, m_0]$ be a Petri net and $L = [S, E, A, I]$ a labeled transition system. L is called the* state space *of N iff*

- $S = \{m \mid m_0 \xrightarrow{*} m\}$,
- $A = T$,
- $I = \{m_0\}$,
- *and* $[m, t, m'] \in E$ *iff* $m \xrightarrow{t} m'$.

In the sequel, we shall frequently refer to concepts closely related to a Petri net (or, more general, a system description) as *structural* while with *behavioral* we shall address concepts closely related to the associated labeled transition system. This is in part justified by the fact that system properties we are interested in are defined over labeled transition systems.

1.2 System properties

In the broadest sense, a property would be any function that assigns values to (labeled transition) systems. Verification would then be the process of determining that value for a given system. For *qualitative* properties, the assigned value is boolean. In that case, we say that a system *holds* the property if the assigned value is *true*. *Quantitative* properties assign numbers. We believe that all relevant system properties belong to one of these classes. At least, if a property assigns, say, a machine code sequence to each system, we would probably replace the term *verification* by something like *compilation*.

In this thesis, we focus on qualitative properties. Quantitative verification requires usually a richer system description (for instance, distributions and rates for occurrence of events) than qualitative properties, and use specific verification techniques (such as Markovian analysis).

We prefer to further narrow our view on properties. We do not naturally experience systems as complete labeled transition systems. What we can see is just sequences of states, altered by events. For instance, we assume that a property like "the labeled transition system graph consists of 25 strongly connected components"is less interesting than "system execution will eventually reach some terminal state". Properties like the first one may serve as a tool for determining other, more interesting properties, but are not relevant as such.

Definition 5 (Path) *Let $L = [S, E, A]$ be a labeled transition system. A state $s \in S$ is a* terminal *state iff there is no $s' \in S$ holding $[s, s'] \in E$. A finite* path *in L is an finite sequence $s_0 s_1 s_2 \ldots s_n$ of states such that, for all $i \in \{0, \ldots, n-1\}$, $[s_i, s_{i+1}] \in E$. An infinite* path *in L is an infinite sequence $s_0 s_1 s_2 \ldots$ such that, for all $i \in \mathbf{N}$, $s_i \in S$, and either $[s_i, s_{i+1}] \in E$ or s_i is terminal and $s_i = s_{i+1}$.*

With the special treatment of terminal states in infinite paths we achieve a uniform handling of terminating and non-terminating executions.

The most simple qualitative observations on a system execution are those on single states. Such observations are formalized as

Definition 6 (Atomic proposition) *An* atomic proposition *$\alpha : S \longrightarrow \mathbf{B}$ assigns truth values to states in a set of states S.*

This definition implies that the assigned value is independent of the position of a state within a labeled transitions system. Furthermore, we assume that α can be evaluated effectively, with negligible time and memory resources, on an explicitly represented state.

Manna and Pnueli [MP92] proposed a language that is widely agreed upon for the specification of path properties. It assumes infinite paths. The language, a *temporal logic*, builds formulas out of a set AP of atomic propositions using the following constructions and meanings:

$\phi \ldots$	is formula iff	and holds for $\pi = s_0 s_1 \ldots$ ($\pi \models \phi$) iff
α	$\alpha \in AP$	$\alpha(s_0)$
$\neg\phi_1$	ϕ_1 is formula	$\pi \not\models \phi_1$
$(\phi_1 \wedge \phi_2)$	ϕ_1, ϕ_2 are formulas	$\pi \models \phi_1$ and $\pi \models \phi_2$
$(\phi_1 \vee \phi_2)$	ϕ_1, ϕ_2 are formulas	$\pi \models \phi_1$ or $\pi \models \phi_2$
$\mathbf{X}\phi_1$	ϕ_1 is formula	$s_1 s_2 s_3 \cdots \models \phi_1$
$\mathbf{F}\phi_1$	ϕ_1 is formula	there is an $i \in \mathbf{N}$ s.t. $s_i s_{i+1} s_{i+2} \cdots \models \phi_1$
$\mathbf{G}\phi_1$	ϕ_1 is formula	for all $i \in \mathbf{N}$, $s_i s_{i+1} s_{i+2} \cdots \models \phi_1$
$\phi_1 \mathbf{U} \phi_2$	ϕ_1 is formula	there is an $i \in \mathbf{N}$ s.t. $s_i s_{i+1} s_{i+2} \cdots \models \phi_2$, and for all j $(0 \leq j < i)$, $s_j s_{j+1} s_{j+2} \cdots \models \phi_1$

If a formula does not use temporal operators (i.e. is a boolean combination of atomic propositions), we call it a *state predicate*. For the intuitive meaning of the non–boolean operators, view a path property from the perspective of the first state. We have consequently **X** —"in the next step", **F** —"eventually", **G** —"globally (always in the future"), and **U** —"until". [MP92] introduces additional operators. One group of these operators provide "strict"versions of the above ones. The strict version **F** , for instance, would be true of ϕ_1 iff $\mathbf{XF}\phi_1$ is true. Another group of operators provides "past"versions (and strict past versions) of the above operators. The past version of **U** could be interpreted as "since", the past version of **G** as "has always held so far", the past version of **F** as "once", and the past version of **X** as "in the previous step". We decided to go with the above fragment of the logic since it is simpler, more closely related to actual verification tool developments, compatible with forthcoming definitions in this section, and, in principle, as expressive as the complete logic.

A labeled transition system TS holds a path property ϕ ($TS \models \phi$) iff

all its paths do. A labeled transition system with initial states holds a path property (we use the notation $TS \models \phi$ for this case, too) iff all paths starting at initial states do. Call the above language *LTL* (linear time temporal logic).

Here are some examples for relevant properties and their expression in LTL (verbal terms refer to atomic propositions):

Variable x will eventually exceed value k:	$\mathbf{F}x > k$;
Pointer p will never show "nil":	$\mathbf{G}p \neq nil$;
A door cannot be opened while train is moving:	locked **U** stopped;
The system will eventually stabilize:	**FG** stable;
The service does not permanently break down:	**GF** service available;
Situation A leads to situation B:	$\mathbf{G}(\neg A \vee \mathbf{F}B)$.

There are properties that are interesting but not expressible in LTL. For instance, a reasonable requirement for an interactive program would be that, at every stage there is a way to quit it (possibly requiring more than one action). This is different from requiring eventual termination. The specification can well be satisfied despite a path that does not ever lead to the *quit* state. It is only important that, at each state on that path, we *could* have gone to the quit state, say, by pressing another button than the one we did.

This specific situation involved actions that can be triggered from outside. This means that nondeterminism in the system (the button to be pressed) can be resolved intensionally. Control theory distinguishes controllable actions (those that can be triggered by purpose) and uncontrollable actions (those that occur spontaneously). Thus, properties like the one above that express *options* rather than actual behavior occur naturally. There is no way to express such properties in LTL since they concern two (or more) paths rather than only a single one (at least, the actual path of system execution, and the alternative path to the "quit"state are involved in the example above).

The adequate model for expressing properties that involve more than one path is the

Definition 7 (Computation tree) *A directed labeled rooted tree is a* computation tree *for a labeled transition system $L = [S, E, A, \{s_0\}]$ with single initial state s_0 iff*

- *every vertex is labeled with a state $s \in S$;*
- *every edge is labeled with an action $a \in A$;*
- *the root vertex is labeled with s_0;*

- *an edge connects two vertices only if the corresponding labels form an event in E;*
- *for every vertex v labeled s, and every event $[s, a, s']$, there is an edge labeled a connecting v with a vertex labeled s'.*

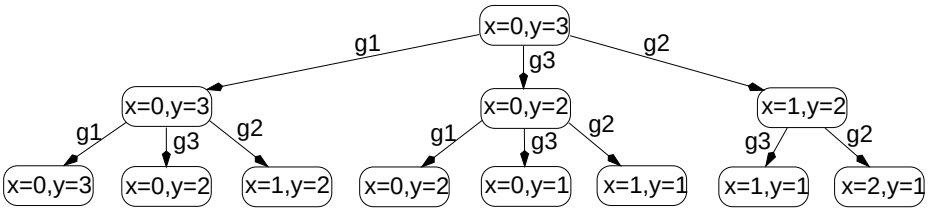

Figure 1.5: First three layers of the computation tree for the system in Fig. 1.1

A labeled transition system with single initial state has, up to tree isomorphism, a unique computation tree. Paths (reduced to labels) in the tree correspond exactly to paths in the original labeled transition system. While LTL regards a system as a plain *set* of possible executions, the computation tree records common initial segments of paths, and therefore the exact state in which nondeterminism is resolved in favor of one or another path.

Emerson and Clarke developed a logic that can be interpreted on computation trees [CE82, CES86b, EH86]. It extends the above list of LTL construction rules by two more operators (called *path quantifiers*).

$\phi \ldots$	is formula iff	and holds for $\pi = s_0 s_1 \ldots$ $(\pi \models \phi)$ iff
$\mathbf{A}\phi_1$	ϕ_1 is formula	for all paths $\pi' = s_0 s'_1 s'_2 \ldots$, $\pi' \models \phi_1$
$\mathbf{E}\phi_1$	ϕ_1 is formula	there is a path $\pi' = s_0 s'_1 s'_2 \ldots$ s.t. $\pi' \models \phi_1$

If a formula $\mathbf{A}\phi_1$ does not hold, there is a path π with $\pi \not\models \phi_1$. Such a path is called *counterexample* for $\mathbf{A}\phi_1$. If a formula $\mathbf{E}\phi_1$ does hold, a path as required by definition is called *witness* for $\mathbf{E}\phi_1$.

This logic is called *CTL** (*computation tree logic*). The property discussed earlier "it is possible to quit a program at any time" can be expressed in CTL* as $\mathbf{GEF}quit$.

The definition of the meaning of the operators $\mathbf{A}$ and $\mathbf{E}$ shows that the values of $\mathbf{A}\phi$ and $\mathbf{E}\phi$ are the same for all paths starting at the same state s_0. Thus the notion of "s_0 satisfies ϕ"$(s_0 \models \phi)$ is justified for such formulas and their boolean combinations. The fragment *CTL* (again: computation

tree logic) of CTL* maximizes the use of a state based view. In CTL, the path operators **X** , **F** , **G** , **U** and path quantifiers **A** , and **E** , occur only as pairs **E X** , **E F** , **E G** , **E** (.**U** .), **A X** , **A F** , **A G** , and **A** (.**U** .). Each of these pairs is treated as a monolithic operator. This way, since atomic propositions do also concern states rather than paths, all values of formulas and subformulas can be assigned to states. This enables efficient verification techniques. The formula **AG EF***quit* expresses the same property as the previous formula.

In the previous section we argued that a more restrictive system description formalism enables a larger number of verification techniques. The same is true for properties. Every verification technique for a large class of properties can obviously be applied to each subclass, while certain subclasses enable additional, more efficient techniques. For instance, for the purely path based logic LTL, or the purely state based CTL, there are techniques that are not available for the tree based CTL*. In the sequel, we consider even smaller classes of properties for which we shall discuss particular verification schemas later on.

A *universal property* is a property that can be expressed in CTL* without using the existential path quantifier and without applying the negation symbol to subformulas that contain path quantifiers (otherwise, one could replace $\mathbf{E}\phi$ by $\neg\mathbf{A}\neg\phi$). The corresponding fragments of CTL and CTL* are called ACTL and ACTL*, resp. It is commonly believed that most properties that express correctness requirements of real-world systems are universal properties. LTL is a subset of ACTL*.

A *reachability property* is a property that can be expressed as $\mathbf{AG}\phi$ with ϕ being a state predicate. This class of properties is in the intersection of CTL and LTL (the definition that a system satisfies a formula iff *all* its paths do, adds an implicit **A** in front of the LTL formula $\mathbf{G}\phi$). Many safety requirements can be expressed in this simple form.

A *home property* is one that can be expressed in CTL as $\mathbf{AG\ EF}\phi$ with ϕ being again a state predicate. Home properties are not expressible in LTL (as we have discussed above). Two other home properties—reversibility (**AG EF***initial*) and transition liveness (**AG EF***t enabled*) have been intensively studied in the Petri net area.

A *goal property* can be expressed in LTL by $\mathbf{F}\phi$ (or, in CTL, as $\mathbf{AF}\phi$) using a state predicate ϕ. It expresses that a ϕ-state will be eventually reached in the system. Goal properties are particularly important for algorithms where

eventual termination or eventual delivery of a result is crucial.

A *stabilization property* is an LTL property expressible as $\mathbf{FG}\phi$ for a state predicate ϕ. Such properties occur in the context of self-stabilizing systems.

An *immortality property* has the from $\mathbf{GF}\phi$ for some state predicate ϕ. It says that, on each path, ϕ-states occur infinitely often. Availability of servers, or proper functioning of communication media, fair arbitration of resources, and many other properties are usually expressed as an immortality property.

Goal, stabilization, and immortality properties are all *liveness properties*, whereas reachability and home properties are *safety properties*. For LTL, where each property can be associated to the set SAT of paths satisfying it, a property is a liveness property iff every finite path can be extended to an infinite path in SAT (i.e. a liveness property can never be qualified as false by just looking at a finite computation). An LTL property is a safety property iff for every infinite path not in SAT there is a finite prefix such that no extension of that prefix is in SAT (i.e., violations of safety properties manifest themselves already after a finite amount of time and are permanent). It is known that every LTL property can be expressed as a conjunction of a safety and a liveness property.

As a purely Petri net specific property, we consider *boundedness*. A place of a Petri net is bounded at a marking m iff there is a number k such for all markings m' reachable from m, $m'(p) \leq k$. A Petri net is bounded iff all its places are. Boundedness of a net is equivalent to having a finite set of reachable markings.

1.3 Fairness

In the framework presented so far, most liveness properties of distributed systems turn out to be false, even if they hold intuitively in the original system. The reason is that a transition system does not implement any assumption about the relative speed of components of the modeled system. It contains therefore infinite executions where some component executes infinitely often while another component does not execute any of its actions at all. Any property that depends on progress in the ignored component will consequently be evaluated to false.

The best way to exclude such kind of behavior without introducing too strong assumptions on the relative speed of components is to introduce fairness. A fairness requirement is a property of paths that distinguishes reason-

able from unreasonable system executions. The distinction of what requirements are "reasonable"or not is left to the modeler of the system. Fairness requirements are therefore to be considered part of the system description. Verification in the presence of fairness requirements means that only those paths of the system are investigated (and within the range of path quantifiers) that meet all fairness requirements.

There are several proposals on how to specify fairness requirements. The two most commonly used concepts are *weak* and *strong* fairness.

A *weak fairness* requirement is specified as a state predicate ϕ. A path is weak fair w.r.t. ϕ iff it satisfies the LTL formula $\mathbf{GF}\phi$. Weak fairness is also referred to as *justice* [MP92] or *progress* [Rei98]. For example, using two weak fairness requirements $x = 0$ and $x = 1$, paths where the value of a boolean variable x remains unchanged forever (from some point on) are excluded. Such a pair of requirements is usually used for variables that represent input from the environment in order to assume ongoing outside activity. For a program counter variable pc, the requirement $pc \neq k$ specifies that the process owning this variable will not rest in program location k forever. The problem of ignored components mentioned above can be solved by a set of weak fairness requirements. The concept of fairness constraints introduced in [BCM+90], coincides with weak fairness with the exception that ϕ can be an arbitrary CTL formula rather than just a state predicate.

A *strong fairness* requirement is specified as a pair of state predicates $[\phi_1, \phi_2]$. A path satisfies this requirement iff the LTL formula $(\mathbf{GF}\phi_1) \longrightarrow (\mathbf{GF}\phi_2)$ is true on that path. In [MP92], strong fairness is called *compassion*. Strong fairness is commonly used to specify reasonable arbitration of resources (for example, by strong fairness requirements of the kind [request, granted]) where the arbitration algorithm itself is not explicitly modeled. In Petri nets, every conflict resolution (the assignment of a token to one of several enabled transitions for executing that transition) is such an arbitration problem. For some systems it is known that they cannot be modeled adequately without strong fairness requirements [WK97].

1.4 Relations between systems

Simulation and bisimulation express a relation between the behavior of different systems and can be used as a tool for proving that a system transformation preserves certain properties. Such transformations are used as part

of the modeling process, or, as a state space reduction technique, in the verification process, to reduce the system complexity thus fighting the state explosion problem.

Definition 8 (Simulation) *Let $\mathcal{T}_1 = [S_1, E_1, A_1]$ and $\mathcal{T}_2 = [S_2, E_2, A_2]$ be labeled transition systems. We say that $\mathcal{T}_2$ simulates $\mathcal{T}_1$ w.r.t. a set P of propositions ($\mathcal{T}_1 \preceq_P \mathcal{T}_2$) if there is a mapping $h : S_1 \longrightarrow S_2$ s.t.*

- *for all $s \in S_1$ and all $p \in P$, $s \models p$ implies $h(s) \models p$; and*
- *for all $[s, s'] \in E_1$, $[h(s), h(s')] \in E_2$.*

If $\mathcal{T}_2$ simulates $\mathcal{T}_1$ then the computation tree of $\mathcal{T}_1$ can be viewed as a subtree of the computation tree for $\mathcal{T}_2$ (with h establishing the connection). Thus, properties that hold for all branches of $\mathcal{T}_2$s computation tree, hold for $\mathcal{T}_1$s computation tree as well. Formally,

Proposition 1 *If $\mathcal{T}_1 \preceq_P \mathcal{T}_2$, and ϕ is a formula in ACTL* using only propositions in P, then $\mathcal{T}_2 \models \phi$ implies $\mathcal{T}_1 \models \phi$.*

Thus, exhibiting a simulation is a useful tool to prove that some transformation of a transition system preserves universal properties.

Definition 9 (Bisimulation) *Let $\mathcal{T}_1 = [S_1, E_1, A_1]$ and $\mathcal{T}_2 = [S_2, E_2, A_2]$ be labeled transition systems. We say that $\mathcal{T}_1$ and $\mathcal{T}_1$ are bisimilar w.r.t. a set P of propositions ($\mathcal{T}_1 \simeq_P \mathcal{T}_2$) if there is a bijective mapping $h : S_1 \longrightarrow S_2$ s.t.*

- *for all $s \in S_1$ and all $p \in P$, $s \models p$ implies $h(s) \models p$; and*
- *for all $[s, s'] \in E_1$, $[h(s), h(s')] \in E_2$.*
- *for all $s \in S_2$ and all $p \in P$, $s \models p$ implies $h^{-1}(s) \models p$; and*
- *for all $[s, s'] \in E_2$, $[h^{-1}(s), h^{-1}(s')] \in E_1$.*

Bisimilar systems relate closely to CTL* formulas. In particular,

Proposition 2 (Properties of bisimulation)

1. *If $\mathcal{T}_1 \simeq_P \mathcal{T}_2$ and ϕ is a formula in CTL* that uses only propositions from P then $\mathcal{T}_1 \models \phi$ if and only if $\mathcal{T}_2 \models \phi$;*

2. *If for all CTL formulas ϕ that use propositions from P only, $\mathcal{T}_1 \models \phi$ implies $\mathcal{T}_2 \models \phi$, then $\mathcal{T}_1 \simeq_P \mathcal{T}_2$.*

That is, bisimilarity is the same as indistinguishability in CTL (or CTL*).

Approximations are simple cases where a simulation relation can be established.

Definition 10 (Approximations) *Let $\mathcal{T}_1 = [S_1, E_1, A_1]$ and $\mathcal{T}_2 = [S_2, E_2, A_2]$ be transition systems. If $S_1 \subseteq S_2$, and $E_1 \subseteq E_2$, we say that S_1 underapproximates S_2, and we say that S_2 overapproximates S_1.*

An overapproximation contains at least the states and transitions of the original system, an underapproximation contains at most the states and transitions of the original system.

Taking the identity as the simulation mapping, we obtain immediately

Proposition 3 (Overapproximation simulates original system) *If $\mathcal{T}_2$ overapproximates $\mathcal{T}_1$ then $\mathcal{T}_1 \preceq_P \mathcal{T}_2$ for all P.*

Thus, all ACTL* formulas that can be verified for some overapproximation, are true of the original system. Despite this nice property, overapproximations by themselves are not a useful tool for explicit state space verification since they are larger than the original system. They can, however, lay the ground for other techniques that rely on some regularity of the transition system, such as symbolic verification, or abstraction techniques.

Chapter 2

Performance evaluation of verification algorithms

The purpose of reduction techniques that we are going to discuss later in this thesis is to save run time and memory resources. In order to compare the various methods, we need to get some impression concerning their time and space performance. This chapter is about finding a suitable method to do this particular task.

2.1 Worst case and average case analysis

Worst case analysis is a well established technique for classifying programs by means of their run time or space behavior. Worst case time (or space) analysis means to find a unary function f such that the run time (or space requirements) for executing the given program on an arbitrary feasible input of size n is less or equal than $f(n)$. Programs with similar time or space bounds form complexity classes.

For most of the techniques studied in this thesis, worst case complexities are well known. Nevertheless, worst case analysis is of no help for comparing them. The reason is that all reduction techniques have a worst case that corresponds to not using that technique at all. Thus, worst case analysis cannot separate any of these techniques, nor can it separate sophisticated reduction techniques from brute force verification.

For worst case analysis, it is sufficient to analyze run time or space for *some* input of size n known to lead to the extremal execution path of the

program under investigation. This is usually much easier than analyzing *all* possible execution paths of a program which would be necessary for average case analysis.

In *average case analysis* , we seek for a function f that bounds the average, not the worst case, consumption of space or time. That is, f must satisfy

$$f(n) \leq \frac{\sum_{i \in I_n} T(i)}{\mathrm{card}(I_n)}$$

where I_n is the set of all feasible inputs of size n, and $T(i)$ is the run time (or space) required for an execution of the studied program on input i.

There are several problems with average case analysis in the domain of this thesis. First, I_n is the set of all systems with a description of size n (bytes of text, for example). It is already a rather complex task to find out its size. The major problem is, however, that we do not have sufficient knowledge of our methods to bound $T(i)$ reasonably. Consequently, average case analysis is not available for our purposes. To complicate matters further, many syntactically feasible descriptions do not represent a reasonable system, that is, one that we expect to occur frequently in practice. In previous sections, we focused on certain classes of systems (finite state, distributed, regularly structured, ...). These attributes are fuzzy in nature and not all of them can be immediately detected in the system description. Structural features of a description may influence time and space more than its size. Thus, average case analysis, even if it were able to separate techniques, would possibly fail to answer the question of how the methods behave in expected cases. The attempt to assign higher weights for the more "reasonable"inputs in the above formula fails due to our fuzzy understanding of what exactly we would like to call "reasonable". The same arguments disqualify probabilistic methods of performance evaluation.

Having seen that systematic approaches to performance evaluation are not available, we need to rely on ad hoc methods. In publications on verification techniques, reports on run time and space behavior on some *particular* set of examples dominate clearly. In order to increase the value of the punctual results, the choice of examples is usually motivated by either having practical (for instance, industrial) background, or by exhibiting features that are claimed to be paradigmatic (thus, more academic in nature) for the targeted class of systems. Among the two classes of examples, those with industrial background appear to have better reputation in the scientific community. A closer look at both options shall, however, lead to the conclusion that

academic examples, called "toy"in the opposing camp, fit much better our particular needs in the context of this thesis.

2.2 Industrial case studies

Several researchers complement their publication of a new verification technique with a report on a case study with industrial background where the new technique has been beneficially applied. "Beneficial"means that a problem could be solved that could not be solved before, or it could be solved using significantly less resources. We should not underestimate the value of such "proof of concept"studies that help us to convince ourselves that research is proceeding into the right direction. Only such case studies can manifest the breakthrough towards practical relevance that all verification technology aims at. However, a complete case study is a complex task and involves various subtasks not directly related to the actual verification algorithm. Among others, success may in large parts rely on

- efficient communication between industrial users and the people doing the case study (especially if the case study is done in academic environment);
- the actual process of generating the tool's input from the actual industrial system, including various abstractions and simplifications;
- experience in recognizing, acquiring, and conducting promising projects.

Concerning the verification tool itself, it is not only the performance of the actual verification algorithm that influences acceptance in practical applications, but also

- a sophisticated user interface;
- support of formalisms and machine interfaces that fit into the technological process in the application area;
- satisfaction of previous users;
- marketing efforts.

Most of the listed factors are uncorrelated to the particular verification technique. They show that the coverage of a technique or a tool in publications or its wide spread routine use may well depend on available work force behind tool development or case studies, and on self-amplifying feedback loops between existing experience and attraction of other users. Thus, the plain number of "success stories"is not a very reliable measure for the efficiency of the involved verification algorithm.

Consider, as an example, the realm of Petri nets, as documented in the various proceedings of the International Conferences on Theory and Application of Petri nets of, say, the last 10 years (documented in various volumes of Springer's Lecture Notes in Computer Science series). The overwhelming majority of case studies reported there involves the tool DesignCPN [Jen92]. Concerning the verification algorithm, most studies use brute force state space generation, if they do exhaustive verification at all (DesignCPN, to date, provides only little support for more sophisticated verification techniques). This tool, in fact, has offered the most mature graphical user interface for years including hierarchy and module concepts. It supports a formalism (coloured Petri nets) that combines the Petri net resource oriented view with more variable oriented ways of thinking (through its annotations in the functional programming language SML), the developing team has launched a number of case studies and could gather a lot of experience thus attracting other users to the tool. All in all, a marvelous job. Nevertheless, it would be ridiculous to conclude that brute force state space generation were a competitive verification technique by any means!

The only success factor of verification tools in routine industrial use that correlates significantly to the verification technique is the choice of formalism. Since many industrial applications are written in some kind of programming language, variable oriented system description formalisms, like guarded command languages, fit easier into that environment than resource oriented ones. As we have already seen, the availability of certain techniques may depend on the underlying system description language. We should, however, not restrict our attention to certain formalisms just because they are compatible with today's industrial environment. Without a demonstration of benefits of some alternative formalism, there would never be a change in the surrounding technologies.

Besides the underlying formalism, there might be differences in the style of user interaction between techniques studied in this thesis and techniques not covered (for instance, theorem proving requires a completely different amount

of user interaction than state space verification), but there is virtually no difference in user interaction between the various fully automated techniques we are going to present.

Now, if it is not the plain number of success stories, can a deeper analysis of these studies enable us to carve out the contribution of the actual verification algorithm? There is still a number of difficulties for using that information for comparative purposes.

First of all, system descriptions used as input of a verification tool are usually incompletely published, be it due to restricted space, be it due to trade secrets of the contributing company. Thus, third parties are generally incapable of repeating the verification job with their own techniques. The published model may already contain abstractions that are done with respect to the targeted verification method. Without the ability of repeating an experiment with third tools, it is hard to distinguish the impact of the verification algorithm from the particular implementation skills, or the surrounding process.

Competitions like the steam boiler study, [ABL96] where several groups were invited to compete on a common study, are too rare to gather reliable data. Industrial examples tend to be more heterogenous than toy examples. Thus, it is harder to address certain performance patterns to structural features of the verified system. The verified systems are often isolated, not giving a satisfying impression for the evolution of performance measure with increasing input size. The verification tool frequently evolves along with the case study. The case study can then take advantage of unreleased, or undocumented features of the tool, or the model can be tuned according to detailed knowledge of the verification engine.

To summarize, industrial case studies may well be suitable to exhibit the feasibility of a complex process involving a verification tool, but they give only a fuzzy impression about actual capabilities of the actual verification algorithms involved.

2.3 Academic benchmarks

The major complaint about using academic toy examples for studying the performance of verification algorithms is that they be significantly different from real-world examples. This is true in the sense that academic examples tend to be simpler. Their descriptions are generally smaller and more regu-

larly structured. They are condensed to exhibit a small number of features. These features, however, are generally agreed upon as paradigmatic for a larger class of systems, including real world ones.

Now, let us discuss to which degree the alleged differences could alter the performance of verification procedures. The size problem can be easily fixed. Academic examples are more likely to be scalable in size than industrial examples. Thus, for some examples we can scale up the problem size to meet current "industrial size"measures. For other examples, we can extrapolate from the available data and estimate the amount of additional resources that would be necessary to solve some problem instance.

Concerning regular structure, we need to refer to forthcoming results. For one of the major reduction techniques, the stubborn set method, there is no reason to believe that regularly structured systems yield better results than heterogeneous ones. The usually more condensed shape of academic systems introduces, in contrary, more dependencies between actions, with a rather negative impact on the performance of the stubborn set method.

For the second widely known technique, symmetries, a regular structure is compulsory. However, the observation that distributed systems frequently consist of a number of equal copies of some component is more or less undisputed. And concerning the symmetry method in real world applications, we would obviously pick a case study where the considered system does feature significant symmetry. So, heterogeneity concerns at most the structure of the components themselves. Comparing two systems of similar size, the homogeneous one is more likely to exhibit more symmetry than the heterogenous one. Now, handling large sets of small symmetries shall turn out to be more difficult than handling small sets of large symmetries. Of course, larger sets of symmetries lead to more condensed state spaces, but the relation between the number of symmetries and the ratio of full versus reduced state space sizes is well understood. Thus, the results obtained from academic examples can very well be used to infer reasonable estimations for larger, more heterogenous examples.

After all, the only opportunity for academic examples to diverge significantly from industrial examples would be that industrial examples share some common patterns that we happened to overlook so far.

The tendency of academic examples to be scalable enables some asymptotic considerations at least on an example by example basis, each example providing a sequence of very similarly shaped systems. The reduction of academic examples to few features can help us to make reasonable guesses about

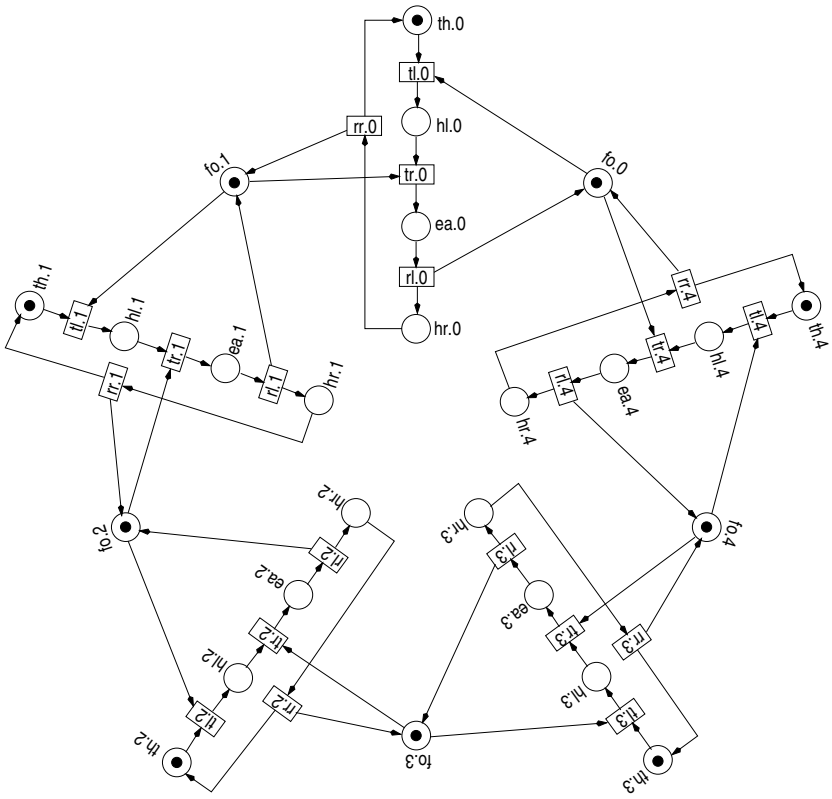

Figure 2.1: Petri net for the five dining philosophers system

the impact of structural patterns on performance. Their simplicity enables third parties to cheaply repeat experiments, to check plausibility of results, or to run them on tools they are less familiar with.

At this stage, we believe that we gathered sufficient evidence to support our decision to illustrate performance of verification techniques by experimental data gathered from a number of scalable academic benchmark systems.

We obtain most of the data from using a single tool that features most of the studies techniques. This way, the implementations share a maximum on common code, thus making run time data even more comparable. The complete source code of our implementations is freely available, and can therefore be easily checked for the absence of hard wired shortcuts ("heuristics") concerning our running examples.

All experimental data, unless explicitly stated otherwise, refer to the tool LoLA [Sch00c], running under the LINUX operating system on a notebook

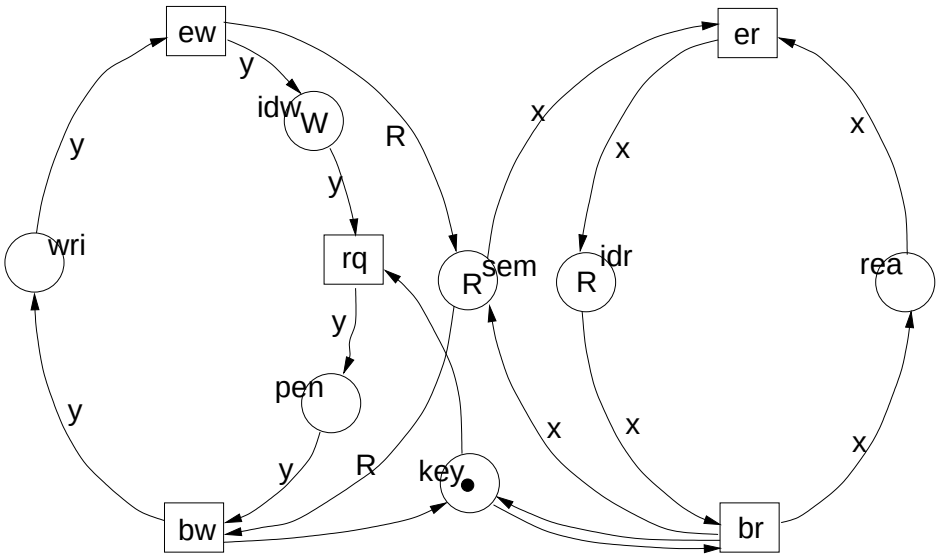

Figure 2.2: Readers and writers in a high level Petri net notation: R and W are sets of process identifiers, an arc annotated with a set means that a transition occurrence removes or produces one element of each set as token on the involved place, evaluations of variables on edges to elements of sets form a firing mode of a transition, and require the production or consumption of the element they evaluate to.

equipped with a 600 MHz Pentium III processor and 256 MBytes RAM. Measured run times are gathered using the shell builtin `time` and comprise the whole process of reading a system description and property to be verified, performing the actual verification, and writing results. This way, run time includes all efforts to speed up verification through pre-processed and statically stored information.

2.4 Running examples

We decided to include only part of the available experimental evidence concerning academic examples into this thesis. We found that a too large number of tables with run-times has a bad effect on readability, and they usually do not give additional insights concerning the principal claims made in the context of the experiments. In order to give some comparatative impres-

sion, we decided to have only two running examples that are probed with every discussed method, and to extend this list punctually wherever these two examples do not exhibit features that are important in some context. Our examples should therefore be understood more as an illustration than a scholarly run time evaluation.

The dining philosophers

This classical example features mutual exclusion between neighbors in a ring, ensured by a set of semaphores. We consider a deadlocking "solution"to the problem.

The example can be adapted to an arbitrary number $n \geq 3$ of processes. It has then $5n$ places, $4n$ transitions, and $3^n - 1$ reachable states. Among the states is exactly one deadlock state, all remaining reachable states are strongly connected in the transition system defined by this Petri net. The dining philosophers system features a rotation kind of symmetry.

Readers and writers

This is another system where semaphores play a key role. They control concurrent access for readers versus mutually exclusive access for writers to a critical section. An additional semaphore (key) avoids conspiration of readers against writers. Without key, there are scenarios where one reader finishes reading only after the next reader starts, thus letting the writing processes starve. A conspiration of writers against readers is still possible if writers stop writing only while the next writing process is already pending.

The system exhibits a dense symmetry group ($R! \cdot W!$ symmetries) and many non–local transitions, due to the simultaneous access of writers to the semaphores. Its full state space comprises of $2^R(W + 1) + W^2$ states.

Part II

State space exploration

All explicit state space exploration techniques are essentially search algorithms that traverse a labeled transition system spanned by a given system description. In the first chapter of this part, we discuss various strategies to traverse a given graph, and some specifics of their implementation in our context. In the second chapter of this part, we show how these search techniques can be employed for the verification of a several classes of system properties.

Chapter 3

Search strategies

We discuss strategies to explore a labeled transition system, or the portion of a system that is reachable from a given state s_0. The latter version, with s_0 being initial state of a system, is the relevant one for explicit state space verification.

The most frequently used search strategy in explicit state space verification is depth first search. For each state s, starting in s_0, depth first search computes the offspring events. Whenever an event leads to a state s' that has not been encountered before, depth first search is immediately launched on s', before continuing exploring the remaining events at s.

A particular run of a depth first search algorithm defines a rooted directed tree that consists of all state as its vertices and a subset of the events of the original transition system as its edges. An edge [s,s'] is a *tree edge* iff the depth first search encounters s' for the first time through exploring events at s. The remaining edges of a directed graph can be partitioned into *forward edges* (edges that lead from ancestors to descendents in the tree), *back edges* (edges that lead from descendants to ancestors in the tree), and *cross edges* (edges that connect different subtrees).

Depth first search is in several respects well suited for explicit state space verification. Besides its easy data structures, the capability of detecting complex graph patterns like strongly connected components, or elementary cycles on-the-fly, is the major advantage of depth first search. Recognizing such patterns is essential for verifying most nontrivial qualitative properties.

The only notable disadvantage of depth first search is its tendency to approach states on paths that are much longer than the respective shortest paths. Thus, depth first search cannot be used for quantitative verification

where result values correspond to minimal path length. Furthermore, paths encountered in depth first search are of limited use as diagnostic information in counterexample scenarios.

3.1 Depth first search and detection of strongly connected components

In this section, we recall the general depth first search algorithm (including recognition of connectivity). This helps us to understand later on, how various reduction techniques modify the general search engine. Then, we discuss how restricted system description formalisms enable more efficient implementation of some features in depth first state space exploration than more general formalisms.

Two states s_1 and s_2 of a labeled transition system are mutually reachable iff $s_1 \xrightarrow{*} s_2$, and $s_2 \xrightarrow{*} s_1$. Mutual reachability is an equivalence relation, and its equivalence classes are called *strongly connected components* (*scc*). In [Tar72], R.E. Tarjan proposed an on-the-fly extension of depth first search that detects all strongly connected components. The algorithm depends on two numbers to be kept for each state. The first number, *dfs*, represents the order in which states have been encountered during depth first search. Whenever search enters a state s for the first time, it assigns a *dfs* value to s that is larger than all *dfs* values assigned so far. The second number, *lowlink*, is initialized to the state's *dfs* number and can be changed upon returning from an exploration of a successor state. After all successors have been explored, *lowlink* contains the smallest *dfs* number among the states that are reachable via arbitrarily many tree edges, followed by at most one back or cross edge to a state *in the same strongly connected component*. Tarjan shows [Tar72] that, after having explored all successors, $dfs(s) = lowlink(s)$ holds iff and only iff s is the root of an scc, i.e. s has the smallest *dfs* number among the nodes of its scc. The other members of that scc are exactly those states that are reachable from the root via tree edges without passing a root state of another component.

The following algorithm explores the set of reachable states from an initial state s_0 and partitions them into strongly connected components. It takes for granted that the system description allows a successive computation of successor states of a given state. The main data structures include a set S

of states (the ones already explored), a subset T of V (the set of states that have been explored but not yet assigned to a scc), the set E of edges, and the *dfs* and *lowlink* numbers, here represented as functions from V into **N**. Outputs are the graph $[S, E]$ (as the final values of the respective variables) and the set of scc (as output in line 22). For better readability, we skip the annotation of edges with actions.

Figure 3.1: General depth first search with scc detection (Tarjan's algorithm)

```
1   var S,T: set of States initial ∅;
2   var E: set of Edges initial ∅;
3   var MaxDfs: N initial 0;
4   var dfs, lowlink: S → N initial ∅;
5   procedure TarjansAlgorithm(current: State)
6   var new : State;
7   begin
8       S := S ∪ {current}; T := T ∪ {current};
9       dfs[current] := lowlink[current] := MaxDfs := MaxDfs + 1;
10      for new in successors(current) do
11          E := E ∪ [current,new];
12          if new ∈ S then
13              if new ∈ T then
14                  lowlink[current] := MIN(lowlink[current],dfs[new]);
15              fi
16          else
17              TarjansAlgorithm(new);
18              lowlink[current] := MIN(lowlink[current],lowlink[new]);
19          fi
20      done
21      if lowlink[current] = dfs[current] then
22          output("scc:",{s | s ∈ T ∧ dfs[s] > dfs[current]});
23          T := T \ {s | s ∈ T ∧ dfs[s] ≥ dfs[current]};
24      fi
25  end.
```

At the top level, call Tarjan's algorithm with the initial state as argument. For understanding the updates of *lowlink*, note that control reaches line 18

for tree edges, and line 14 for other edges. Furthermore, a forward edge does not contribute to the minimum since descendants have always larger *dfs* values than their ancestors.

The time and space complexity of Tarjan's algorithm is $O(|S|+|E|)$, with S and E being the final values of these variables, that is, the number of nodes and edges of the constructed graph. The critical resource in explicit state space verification is space. Thus, we would like to discuss space issues more thoroughly.

It may be the case that we are rather interested in elementary cycles than in strongly connected components. Depth first search is capable of yielding relevant information. Observe that for each elementary cycle reachable from an initial state of a labeled transition system, there must be a state s that would be encountered first. Since all the other states in the cycle are reachable from s, they are descendants of s in the search tree. Thus, there is a back edge from the predecessor of s (in the cycle) to s. Consequently, the set of all states that are entered via a back edge during depth first search, intersects with every elementary cycle of the reachable portion of a labeled transition system. This set can be computed on-the-fly, since back edges correspond to proceeding to successor states that are part of the search search at that time. Furthermore, in that particular moment the portion of the stack between the source and the destination of that back edge form one particular elementary cycle (though there may be cycles that do not exhibit themselves this way, as shown in Fig. 3.2).

3.2 Implementation issues

Functionally, data structures can be divided into the following units (physically, units can overlap in actual implementations):

- The *actual system description* in a form that enables an efficient retrieval of actions enabled in a given state, as well as an efficient computation of successor states; for reduced state space generation, additional requirements may arise;

- A *stack* for implementing the recursion in line 17; stacked information includes the current state and an iteration index for the loop starting in line 10.

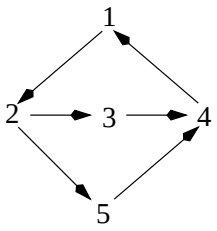

Figure 3.2: The states of the depicted graph are labeled by their respective dfs numbers thus defining the evolution of a particular run of depth first search. When search explores the back edge $[4, 1]$ from current state 4, the stack starting from 1 consists of 1,2,3,4—one of the two elementary cycles. For the second cycle—1,2,4,5— 4 has already left the stack before 5 is explored, so this cycle does never manifest itself through a stack portion. Nevertheless, one member of this cycle (state 1) can be recognized through the back edge $[4, 1]$.

- A *search structure* that implements the set S in a way that containment queries can be handled efficiently and attached information (*dfs*, *lowlink*, connected edges, values of atomic propositions, ...) can be retrieved and updated;
- A *component stack* implementing T;
- The actual *transition system graph*, with traversal capabilities depending on the verification problem.

In the design of a data structure for the *system description*, we are faced with a space/time tradeoff. For time efficiency, information required for enabling checks or successor computations should be preprocessed such that it can be directly accessed. For space efficiency, we would re-compute that information every time we need it, out of a more compact system representation. While, on one hand, explicit information may consume just the space that would have lead to successful completion of a search (as exhibited, for instance, in [MÖ1] for arguing in favor of a highly implicit high level Petri net representation as opposed to explicit low level nets), too implicit information may lead to a prohibitively slow algorithm. Though space is the critical resource in general, data structures for system descriptions are generally no major contributors to space complexity. So, there *is* some room left here to trade space for time.

An efficient implementation of enabling checks and successor computations should at least take advantage of the fact that actions tend to be *local*. Between a state and one of its successors, only few components are different, and for only few actions enabledness depends on those components. Which components are changed, and which actions depend on them, is usually obvious from the system description: a guarded command changes the variables mentioned in its assignments, and its enabledness depends on the variables mentioned in the guard. A Petri net transition t changes places in ${}^\bullet t \cup t^\bullet$ and its enabledness depends on ${}^\bullet t$. Local actions in composed systems depend on and change only local states. Thus, the set of enabled actions can be computed incrementally by checking, after a successor computation, only those actions for enabledness that depend on changed variables. For monotonous enabling conditions, like in the case of Petri nets, the number of checks can be reduced even further: a previously enabled transition needs to be checked only if tokens have been removed from pre-places, and a previously disabled transition needs to be checked for enabledness only if tokens have been produced on the pre-places. Information about actions to be re-checked after an occurrence of an(other) action can be explicitly recorded as part of the system description and is a case where the time gains are worth the (moderate) additional space requirement.

A brute force implementation of the search *stack*, would in turn consume large amounts of space and time. The stack corresponds to a cycle free path in the transition system. Such a path can become extremely long during depth first search. Thus, at least the space consuming stack of current states should be stored only implicitly. In several implementations, the current state information on the recursion stack is realized as a reference (pointer) into the search structure from which the state can be restored.

If all actions are *invertible*, and the reverse of actions can be computed efficiently (for Petri nets, these assumptions are true), information about the current state can be completely reconstructed out of the involved action. The action that leads to the successor state is determined by the loop index (line 10), now the only information that needs to be kept on the stack. When backtracking from a subsequent call of Tarjan's algorithm, the current state can be reconstructed by executing the reverse action at the current state used by the subroutine. Thus, only one single, global variable containing the current state is sufficient for depth first search. As an additional advantage, successor state computation and backtracking can be implemented as an

actual modification of the global current state variable, with no need to copy components that are left unchanged by the action. This means that the time complexity of enabledness checks, action occurrences, and backtracking from subsequent states does not depend on the number of components of the state vector, but only on the average number of components an action is directly related to. Due to locality, this yields a tremendous efficiency gain.

In general, it is convenient to assume that, for space efficiency reasons, the search structure for S overlaps with the actual implementation of the set S. That is, in addition to procedures to search and insert elements, we need a way to retrieve an actual element of S via a reference. A reference is a small piece of data (usually a pointer) from which a state (the assignment to all state variables, or marking on all places) can be reconstructed efficiently. We want to use references on the stack (unless we have invertible actions), and in the explicit representation of edges in the transition system in order to avoid space consuming multiple representations of elements of S.

Older verification tools (e.g. EMC [CES86b]) store S as a linked list of assignment vectors, yielding an efficient insertion procedure, simple pointers as reference to elements, but a rather expensive search procedure. Search can be speeded up by putting a hash structure or a search tree on top of the list. Alternatively, the list can be replaced by a tree where each path in the tree corresponds to a state, searching corresponds to traversing the tree from top to down, inserting to adding a new branch, and references to leaf nodes from which the referenced state can be retrieved by traversing the tree from bottom to top. Fig. 3.3 depicts a search tree as used in our implementation.

If we need to access the search structure via references, our freedom to implement the search structure is limited (compared to a data structure where search and insert are the only operations to be implemented). In forthcoming chapters, we shall argue that all verification techniques we are going to present, and most reduction techniques can be implemented without relying on references to stored states (they only need to investigate the respective current state). If, furthermore, the stack can be implemented without references, for instance due to having invertible actions, we get a lot of freedom to implement the search structure. For instance, we can transform a state into smaller representation yet identifying it uniquely, and store only the representation—even if it is hard or impossible to reproduce the actual state out of that representation. In Ch. 8, we are going to propose such a "fingerprint"transformation of states. In order to establish compatibility between

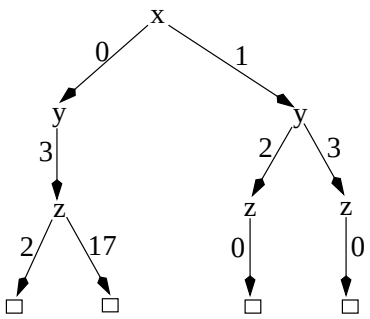

Figure 3.3: A decision tree storing a set of states (assignments to the state variables x, y, and z). The states represented by this particular data structure are $(x = 0, y = 3, z = 2)$, $(x, 0, y = 3, z = 17)$, $(x = 1, y = 2, z = 0)$, and $(x = 1, y = 3, z = 0)$. If the edges of the decision tree can be traversed backwards, the dummy leaf objects can be used as references to states.

fingerprint transformations and other verification techniques, it is therefore important to try to avoid the use of references to states wherever possible. This is why state reference considerations are a recurring issue in forthcoming discussions sections.

The *component stack* T does not require extra space. T is always a subset of S, and we can represent it as a bit in the representation of elements of S. For instance, the value -1 for *lowlink* can be used to mark explored states not in T (the *lowlink* value is irrelevant for elements outside T).

Concerning a space consuming explicit representation of the actual *transition system*, all future considerations will focus on avoiding such a representation as far as possible. It shall turn out, that in most cases all necessary information for verification problems can be gathered during the actual depth first search, with no need for extra navigation through the transition system.

3.3 Breadth first search

Unlike depth first search where the last recently encountered state is always the first to be explored further, breadth first search processes states in the order they have been encountered. Thus, the distance of states to be explored from the initial state is monotonously increasing. This, in turn, means that among all states holding some property, the first state encountered is the

one with minimum distance from the initial state. This phenomenon makes breadth first search attractive for purposes where paths from the initial state to distinguished goal states are processed further and should therefore be as small as possible.

Unfortunately, it is harder in breadth first search to detect cycles or strongly connected components than it is in depth first search. The reason is that components can, in general, not be localized as subtrees in the breadth first search tree (which is the case in depth first search and fundamental for Tarjan's algorithm). Cycles may be scattered as well if there are multiple entries into them from states with similar distance from the initial state.

In the previous section, we observed that two subsequently considered states in depth first search can be transformed into each other by letting invertible actions occur forward or backward. We claimed that this kind of transformation is cheaper than copying states, and leads to more condensed implementations of the search structure (through fingerprints). In breadth first search, consecutively processed states are in general not immediately related through a single action. They tend do have, though, immediate or almost immediate common ancestors. Thus, we can still transform a currently processed state into the next state to be processed by a short sequence of backward and forward actions. This way, all the advantages discussed in the section on depth first search (fingerprint implementation of the search structure, locality of state transformation, incremental enabledness test) apply, in a slightly weaker form, for breadth first search. Implementation of breadth first search using backward and forward actions is fairly easy: We can simply use depth first search, applying a depth restriction k (initially 1): for states in distance k from the initial state (in terms of the depth first search tree), we do not recursively explore their successors but backtrack immediately. Upon completion of such a depth first search, we increment k and launch a new depth first search, and so on, until no new states are encountered during a full iteration. It is well known that, despite multiple encounter of many states, this implementation does not add principal complexity to breadth first search. This fact, together with the gains from locality, incremental computations, and a more efficient search structure, lets us recommend incremental depth first search as implementation of breadth first search.

3.4 Distributed search

If the state space is constructed by a cluster of workstations, neither depth first nor breadth first search is applicable since otherwise it would be impossible to use the parallel computing power for exploring states in parallel. Parallel computing power is needed for compensating the overhead caused by communication between the workstations, at least partly. There are two objectives for using clusters of workstations in state space verification. First, there are more resources available for storing states, so larger state spaces can be explored without running out of memory. Second, the parallel processing of states may speed up computation (if this speed-up is not compensated by the communications overhead), so the overall run time of state space generation shrinks. The second objective was central for the distributed algorithm used in the Murϕ tool [SD97]. They used a time-expensive procedure to compress states before storing them, and distributed state space generation helped them to compute many of these compressions in parallel thus obtaining significant speed-ups. They did not, however, address the storage objective. They used a hash function to determine a workstation where the state in question should be stored and explored. Though they reported that they obtained an even distribution of states among the workstations in their examples, it is not clear whether this observation would be equally true if we dealt with hundreds of workstations rather than with tens. In fact, a hash function usually does not exploit particular structural knowledge about the system, so the best behavior we can expect is a random-like (uniformly distributed) assignment of hash values. Unfortunately, in such a stochastical setting, the expected number of states at the moment when the first machine runs out of memory grows much slower than the number of participating machines. It is not easy to change a hash function once a state space generation is running, so there are only few possibilities to react online to an unbalanced distribution of the state space.

In the sequel, we propose a different distribution scheme, with main emphasis on the full exploitation of memory on all involved workstations. Our data structure offers interesting opportunities for load balancing on-the-fly. Furthermore, we do not want to rely on an expensive local compression procedures in order to decrease the communication bandwidth. On the other hand, we put less emphasis on the run time issue. We must admit that we have only limited experience with our data structures at this time. We must therefore leave final conclusions concerning performance of this technique to

future research. We find it nevertheless very promising, particularly for its capabilities to control the load on all participating machines throughout the whole process of state space exploration.

Our data structure can be seen as a distributed implementation of a binary decision tree. We assume that the participating workstations are fully interconnected via point to point message passing. We assume furthermore that the (asynchronous) message passing protocol meets the following specifications:

- Messages cannot get lost nor corrupted;
- The order of arrival can be different from the order of sending;

This specification is more strict than the standard protocol UDP, where messages can get lost (and which is a broadcast protocol), but less strict than the widely used TCP/IP, where the order of messages is preserved. More relaxed specifications enable more efficient implementations. In LoLA, we built a simple protocol on top of UDP that works more efficiently than TCP/IP and meets exactly the required specification.

A binary decision tree stores a set of boolean vectors of dimension n. A full decision tree is a complete binary tree of depth n where the leaves are labeled with true or false. We attach a depth to each vertex in the tree, starting with depth 0 for the root, and ending in depth n for the leaves, Each boolean vector b corresponds to a particular leaf, in other words a path, in the tree. This leaf can be reached from the root by taking, at depth k, the left successor if $b[k] = \text{false}$, and the right successor if $b[k] = \text{true}$. A decision tree stores the set of vectors (states) the corresponding leaf of which is labeled true. In an actual binary decision tree, only the vertices that can reach a leaf labeled true are present. The size of an actual decision tree is proportional to the size of the represented set of states. In the sequel, we consider actual decision trees only.

We distinguish a logical and a physical distribution of an actual binary decision tree (in the sequel, we call an actual binary decision tree just decision tree). Let $\Pi = \{P_1, \ldots, P_n\}$ be the processes participating in state space generation. Assume that every vertex in the original decision tree is labeled (coloured) by an element of Π. We say that process P_i *owns a subtree* of the decision tree iff the root of that subtree is coloured P_i. P_i *owns a state* iff the leaf representing that state is coloured P_i. The process owning a state is responsible for computing that state's successors. The owner of a subtree

is responsible for storing (and then owning) states belonging to that tree, or forwarding them to other processes if those processes own sub-subtrees. The owner of a subtree can shift parts of its realm to other processes. The owner of the root of the decision tree is called *master*. The actual distribution is obtained by starting a local search in the master process, then shifting repeatedly subtrees to other processes.

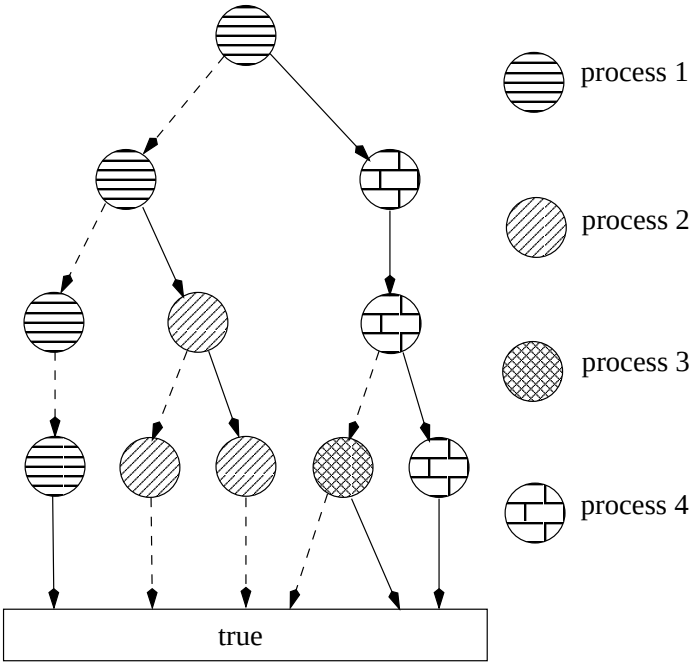

Figure 3.4: A logically distributed binary decision tree, involving 4 processes. Dashed lines are assumed to be labeled 0, solid lines are labeled 1. Process 1 is master and owns state (0,0,0,1). Process 2 owns states (0,1,0,0) and (0,1,1,0). Process 3 owns (1,1,0,0) and (1,1,0,1). Process 4 owns (1,1,1,1).

We refer to the *physical* data structure as the local data structures present in each process, collectively implementing the logical data structure.

Every process holds as its physical data structure a structure similar to a binary decision tree. As the only difference, there may be inner vertices without successors. As in the logical data structure, every vertex is coloured with a process identifier. However, information stored in each process is only partial. The local tree of P_i consists of the following elements contained in the logical data structure:

- all vertices coloured P_i;
- all vertices on paths that lead from the root to vertices coloured P_i;
- all immediate successors of vertices coloured P_i (if such successor is present in the logical data structure);

The size of a physical data structure is proportional to the number of states owned by P_i.

The colors of vertices not coloured P_i are not necessarily the same in the local tree of P_i as they are in the logical data structure. This reflects the fact that a process has only partial knowledge of the evolution of the logical data structure in other processes. The color of a vertex v coloured $P_j \neq P_i$ in the logical data structure, is P_k iff P_k is the color of the deepest immediate successor of a node coloured P_i on the path from v back to the root in the logical data structure (P_k is the root itself if no nodes coloured P_i are on the path from v back to the root). See Fig. 3.5. In other words, P_i "knows"only the immediate successors of its own vertices (besides the master), and these immediate successors manage the whole subtree they own.

It is immediately clear that the logical data structure can be retrieved from the collection of all physical data structures in the participating processes.

The main operation to be implemented on our data structure is search-and-insert. Given a state that has been computed by some process, we need to find out whether the state is already present in the logical data structure. If not, we need to add a new branch to the logical decision tree, assign colors, and let the process that becomes owner of the new state, compute its successors. For the process issuing the search-and-insert request, it is only important whether it is obliged to compute that state's successor or not. There are two possible reason why a process does not need to compute successors—if the state already exists, or if some other process is responsible for computing those successors. Which of these two cases applies is irrelevant for the continuation of the process issuing the request.

Search-and-insert is implemented by two types of actions in processes. A SEARCH action is parameterized with a state. It can be triggered locally, through the computation of a state in the local state space exploration, or by a message from another process. SEARCH consists of traversing the logical data structure from the root down, according to the path defined by the given state. If that path does not end in a vertex owned by the executing

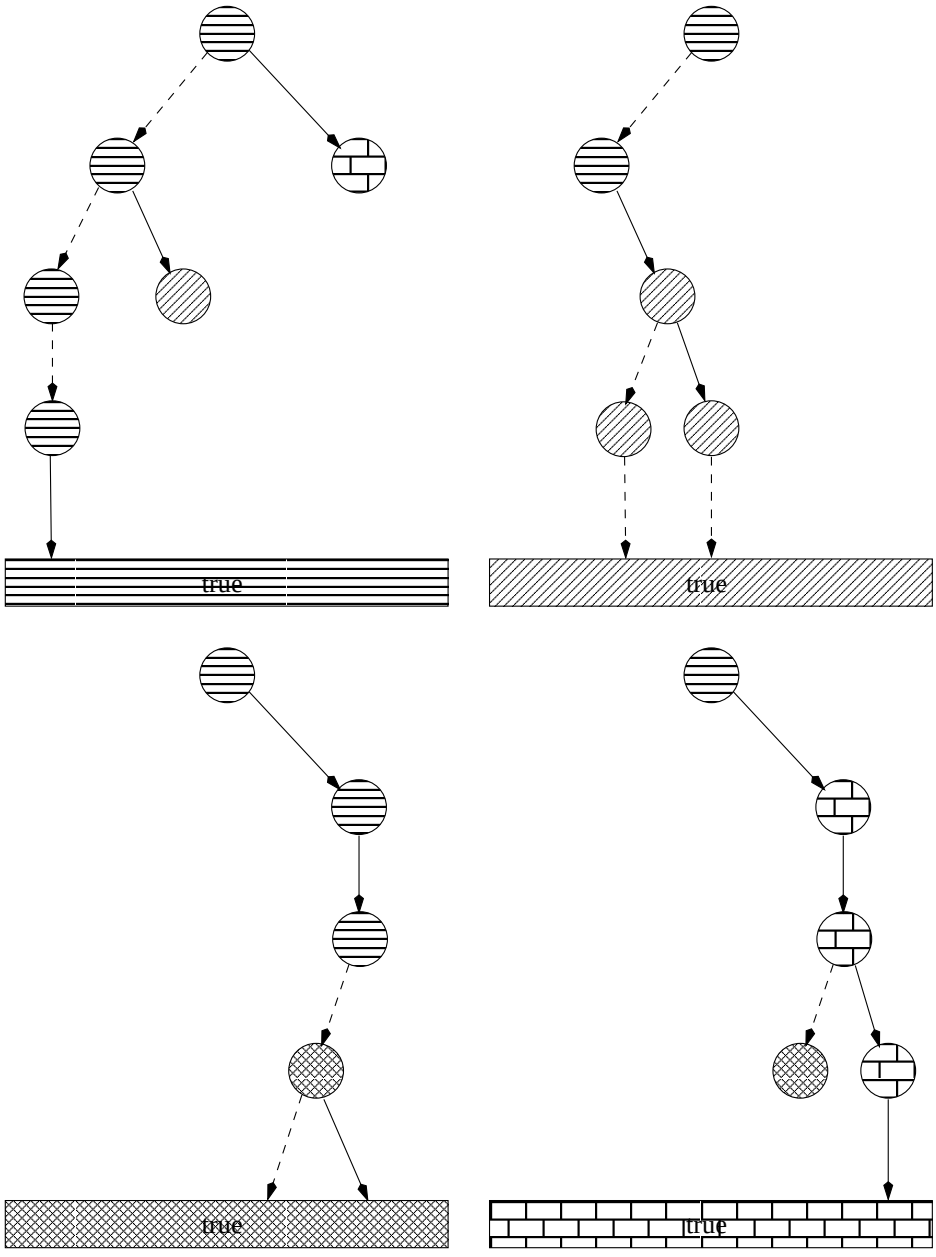

Figure 3.5: Local data structures of the four processes corresponding to the logical data structure above.

process, new messages are triggered. If it does end in a vertex owned by the executing process which is not a leaf, the state is inserted by appending the corresponding branch to the decision tree, or by delegating it to another process, using a specific DELEGATE message.

A DELEGATE action is parameterized with a state and a depth (a number k). It requests the process to assume ownership of the given state, and for the whole subtree defined by that state's prefix up to depth k. DELEGATE is the tool for shifting subtrees to other processes. DELEGATE can be sent only by processes owning the immediate predecessor of the shifted subtree.

We consider first the implementation of a SEARCH action in P_i, triggered by a message or a request in the local search procedure. It starts with a traversal of the local tree, according to the path defined by the given state. This traversal ends in a vertex at some depth k.

Case 1: If $k = n$ (the size of the state), the state exists already, so we do not need to trigger computation of its successors. No modification of any search structure applies, and no further message is issued.

Case 2: If $k < n$, and the final vertex on the traversed path is coloured P_i, then the state is new. In this case, there are two options. First, P_i can decide to take this state. In this case, it adds the remaining vertices for representing the state in its own physical data structure. If the search request is the result of a search message, it adds the state to the queue of states to be locally explored. If the search request has local origin, the search procedure continues exploring successors of the considered state. Logically, we have added a branch in P_i's color to a vertex in P_i's color, so no physical data structures of other processes need to be involved. Second, instead of storing the state locally, P_i can decide to delegate the state, together with some subtree, to another process. In this case, it sends a DELEGATE message to a process $P_j \neq P_i$, using the current state as parameter as well as a number k that is larger than the depth k of the last traversed vertex. Locally, P_i inserts vertices corresponding to all components of the state up to depth k. It colors the new vertex at depth k with P_j, and all remaining inserted vertices P_i. In this case, local search does not compute successors since the receiver of the DELEGATE message becomes responsible for executing this task. The option to pass a subtree to another process is the only available one if a process has run out of memory.

Case 3: If $k < n$, the last vertex on the traversed path is P_j, different from P_i, and the whole traversed path does not contain any vertices labeled P_i

then we trigger a SEARCH message with the given state to P_j. In this case, P_j is the master. Locally, successors of the given state are not computed. The master owns the state or is obliged to dispatch the state to a responsible process.

Case 4: If $k < n$, the last vertex on the traversed path is P_j, different from P_i, and the traversed path contains vertices labeled P_i then we trigger a DELEGATE message with the given state to P_j. As depth for DELEGATE, we use the depth of the immediate successor of the deepest node coloured P_i on the traversed path. Locally, successors of the given state are not computed.

The fourth case is the most involved one. The fact that we send a DELEGATE message rather than a SEARCH message may be surprising. The reason for this choice is that we rely on a message passing protocol that allows messages to overtake. The local data structure of P_i means that, at some point in the past, P_i has delegated a subtree to P_j, the same subtree as coded in the new DELEGATE message. If we sent a simple SEARCH message in case 4, the new SEARCH message could overtake the original DELEGATE message. In that case, P_j could deal with the message differently, for instance delegate some subtree to a third process, or back to P_i. This would result in inconsistencies of the physical data structures. Using the DELEGATE message, P_j becomes responsible for the same subtree as with the original DELEGATE message, so the order of arrival of the two DELEGATE messages does not matter. All we need to take care of is to let a process tolerate receipt of multiple DELEGATEs for one and the same subtree.

The SEARCH action, seen as distributed over several processes, terminates in every case. In cases 1 and 2, it terminates locally. In cases 3 and 4, we can assign a termination function. Let $t = 0$ for a sent SEARCH message, and t be the contained depth for a DELEGATE message. This number is strictly increasing with every passed message. In case 3, it is increasing since $t = 0$ in the sending process (by case assumption), and $t > 0$ in any subsequent message sent by the receiving process which is the master and owns at least the root. For case 4, the receiving process owns at least the immediate successor of the last node owned by the sending process, so t is increasing as well, given that execution of DELEGATE terminates.

Execution of a DELEGATE action in P_i starts as well with a local traversal, according to the state parameter.

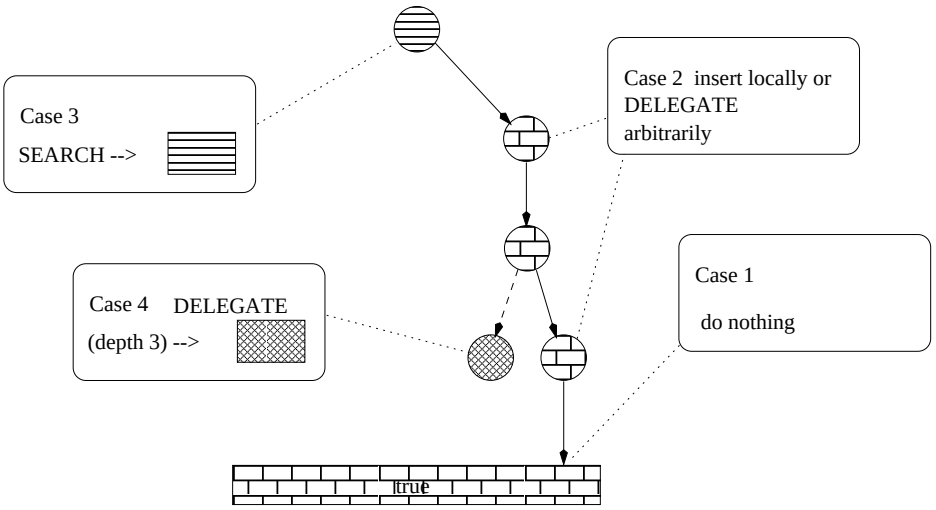

Figure 3.6: Reaction of a process to a SEARCH request, depending on the deepest matching vertices of the searched state

Case 1. If this traversal ends earlier than the depth parameter k, the last vertex in this traversal must be coloured with an identifier different from P_i. Having an own vertex without successor means that in our SEARCH/DELEGATE protocol only P_i itself can decide the logical color of the immediate successor, in contradiction to the fact that another process claims ownership for the vertex according to the given state at depth $k-1$. In this case, P_i must accept the new subtree. It inserts at least the remaining vertices down to depth k, colors the vertex at depth k P_i, and the remaining inserted vertices with the color of the last traversed vertex. Then P_i decides whether to accept the state itself or to delegate it further, with some depth greater than k. The remainder of this procedure works exactly like case 2 of a SEARCH action. The requirement that P_i needs to accept the subtree is necessary to avoid data structure inconsistencies since the sending process has already marked P_i as the owner of that subtree and does not wait for acknowledgment. Additionally, the fact that a process can respond to a DELEGATE message at most with a DELEGATE message containing a larger depth parameter, guarantees eventual termination.

Case 2. If the traversal ends at a depth $\geq k$ (the depth parameter), then it can be treated as a SEARCH. In that case, P_i is already aware of its ownership of the delegated subtree (otherwise, search traversal could not

have ended beneath the passed depth).

It is easy to see that, for every state, the sequence of SEARCH and DELEGATE messages terminates, and the last process involved in that sequence finally owns the state (and is responsible for computing successors).

Example. Assume a distributed situation as depicted in Fig. 3.5. We play first a scenario where the diagonal process produces state (1,1,0,0)—obviously owned by the crossed process in the logical data structures (remember that dashed lines represent 0 and solid lines 1). The local traversal in the diagonal process ends in depth 0. Case 3 applies, and a SEARCH message is sent to the horizontal process (the master). The master conducts a local traversal that terminates in the bricked vertex (the solid line matches the first component of the vector. The master issues a DELEGATE message with depth 1 to the bricked process. Local traversal in the bricked process ends in the crossed vertex (the first 3 components of (1,1,0,0) match). Consequently, a DELEGATE with depth 3 is sent to the crossed process. The crossed process traverses the state completely. None of the processes needs to compute successors of (1,1,0,0) unless that state is still pending in the crossed processes queue.

As a second scenario, assume that the diagonal process initiates a search for (1,0,0,0). As before, it sends a SEARCH message to the master which forwards a DELEGATE with depth one to the bricked process. Local search in the bricked process ends at depth 1 (only the first component of (1,0,0,0) matches). It is now up to the bricked process to decide whether to accept the state. If it does, it inserts a chain representing the remainder of (1,0,0,0) to its local tree and adds (1,0,0,0) to its queue. If not, it issues a DELEGATE, say to the diagonal process with a depth greater than 1, say, depth 3. The diagonal process conducts another local traversal, still ending at depth 0. Case 1 of the DELEGATE protocol applies, and the diagonal needs to insert a new subtree. Assuming that the diagonal process accepts the state, the local data structures of the diagonal and the bricked processes now look as depicted in Fig. 3.7. The physical data structures of the remaining processes remain unchanged.

At every stage, a process that is about to store a state locally can decide to delegate the state instead to another process. Only delegated subtrees must be stored unconditionally. A process delegates states (and with it whole subtrees) at least if it is about to run out of memory. It can choose a new host arbitrarily. However, processes that are already out of memory should

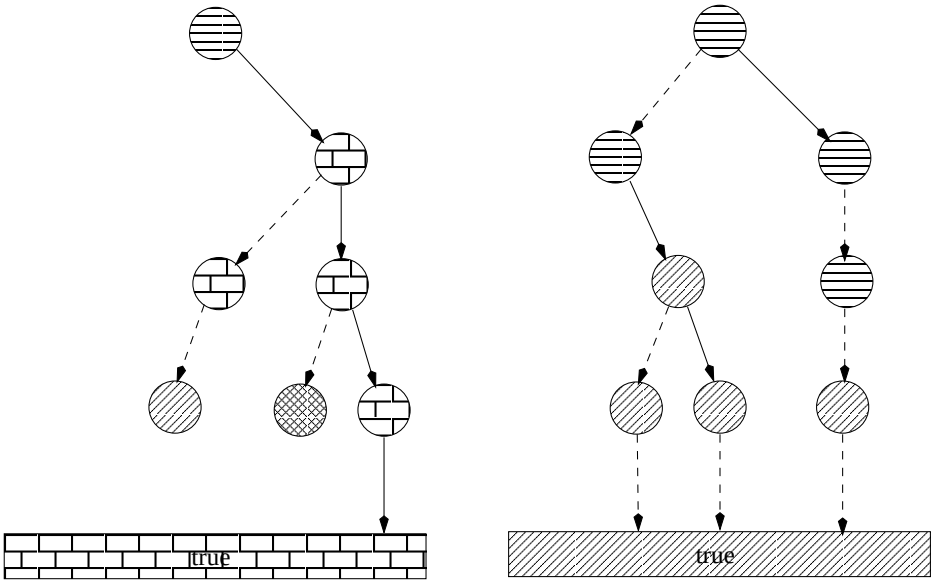

Figure 3.7: New local data structures after delegation of state (1,0,0,0) from the bricked process to the diagonal process, starting from the situation in Fig. 3.5. It is a feature and not a bug that the new chain in the diagonal process is mostly coloured horizontally. This color means only that corresponding search requests are sent to the master who forwards them to the bricked process as the real owner of that subtree.

be exempt. This way, a process out of memory is only as long receiving delegated subtrees as it takes other processes to recognize that fact (via, say, broadcast messages containing a processes state sent on a regular basis by every process). In other words, a process can exhaust its local memory, up to a small amount of memory for accepting pending delegated subtrees.

It is also possible to delegate subtrees before running out of memory. Through delegation, the master who begins state space exploration locally, starts to distribute load after having computed a first initial decision tree that consists of a small number of states. Delegation can be used to shift load away from overloaded processes. The decision whether or not to delegate states, and where, is made exclusively by the delegating process, while state space exploration is running. So, in difference to hash techniques, available information about actual load situation in all processes can influence that decision.

Using this data structure, actual computation of successor states differs only little from sequential depth first search. As a minor difference, search and insert are now a monolithic procedure. Other than this there is no change inside a running depth first search since we have seen that a process does not need to distinguish between an existing state and a state it is not responsible for. The main difference is that there is now a queue of pending "initial"states. Whenever a depth first search is completed, a new state is taken from that queue, and a new depth first search is launched. The queue is filled with states that are received through DELEGATE and SEARCH messages from other processes where this process assumes ownership. Through a specific termination protocol, distributed search ends as soon as all processes have finished their searches and have empty queues.

The proposed algorithm is able to compute the set of reachable states. So far, we do not have an efficient solution for storing events, or to compute strongly connected components. Thus, our distributed search algorithm can, at this time, be used only for properties that can be evaluated from the plain set of reachable states, such as reachability properties.

Whether the proposed data structure, hash values, or other techniques are used for distribution, distributed state space search has a major time disadvantage, compared with local state space exploration. Since states are frequently shifted between processes, the advantages of incremental computations of successor states, set of enabled actions etc. are lost. Since our implementation of local state space exploration uses all these techniques, we cannot exhibit experiments where distributed state space exploration runs faster than local exploration. We can, however, show that we are able to solve larger problems than with local search. We found that network bandwidth of a usual local area network with mixed 10MB and 100MB Ethernet connections is sufficient to satisfy the communication requirements with reasonable delays. Compared with distribution based on hash functions, we believe that our distribution scheme requires less states to be shifted to other processes. We argue that a successor state is equal to the original state in most components. Thus, in more cases than in a random setting, the successor state falls into the same subtree (have a common prefix of considerable length) as the original state thus not requiring interprocess communication.

As an example, the state space of a 1000 philosophers system reduced by basic stubborn sets (see Ch. 5) has 2,997,002 states and 3,997,000 edges. It cannot be verified on the machine used for all other experimental results. On a network of 15 SUN workstations, each single one much slower and equipped

with less memory, the state space could be constructed within less than 5 hours. Thereby, several of the 15 involved machines reached their memory limits.

Chapter 4

Explicit state space verification

So far, we have considered different search algorithms in isolation. We are now going to study how search algorithms are used in the verification of particular classes of properties.

4.1 Reachability properties

The question whether a system satisfies $\mathbf{AG}\phi$ reduces to the question whether a state is reachable from the initial state that satisfies $\neg\phi$. ϕ itself is a boolean combination of propositions, thus evaluating ϕ in a particular state can be done efficiently. To add even more efficiency to that task, we can evaluate ϕ incrementally, that is using its value in some state for computing it in a successor (or predecessor) state in the search tree, as follows. Consider the syntax tree of the formula, i.e. a tree with propositions as leaves and boolean operators as inner nodes. Knowing that an action (or inverse action) changes only few components of the state vector, and assuming further that atomic propositions tend to depend on only one or two components, we can start re-evaluating all those propositions that depend on state components changed by the executed actions. These changes concern leaves of the syntax tree. Only if the value of a subformula changes, we need to re-evaluate the parent subformula. For a negation parent, we can propagate the change immediately further. For conjunction and disjunction parents, we can update the *number* of satisfied subformulas of that conjunction or disjunction. From that number we can deduce whether or not the value of the disjunction or conjunction changes. This way, many formula value updates require only

local computation which makes it as efficient as incremental computation of current state and enabledness.

For evaluating reachability queries, information about strongly connected components of the state space is irrelevant. Hence, we do not need data structures for *dfs*, *lowlink*, or T, and we can choose between depth first and breadth first search. Furthermore, every reachable state can be checked for ϕ when it is the current state to be explored. Thus, there is no need to traverse the transition system and we do not need to store E.

The depth first version of the new algorithm can be obtained from Tarjan's algorithm (Fig.3.1) by removing lines 2-4,9,11,12-15,18,21-24, and the parts concerning T in lines 1 and 8 while inserting a data structure for ϕ, and formula updates before and after the recursive call in line 17 of Tarjan's algorithm.

Figure 4.1: Depth first reachability verification; $\mathbf{AG}\phi$ holds iff procedure does not exit with *false*;

```
1       var S: set of States initial ∅;
2       procedure Reachability(current: State)
3       var new : State;
4       begin
5           S := S ∪ {current};
6           if current ⊨ ¬φ then
7               exit false;
8           fi
9           for new in successors(current) do
10              if new ∉ S then
11                  update formula (forward,φ);
12                  Reachability(new);
13                  update formula (backward,φ);
14              fi
15          done
17      end.
```

The data structure for S can be condensed to serve the operations insertion and containment test. Only for noninvertible actions it would be necessary to restore current states out of the data structure for S. For invertible

actions (using the incremental update of current states), it is sufficient to store a unique fingerprint of a state permanently.

The reachability algorithm illustrates the principle of on-the-fly verification. The verified property is evaluated at the same time the labeled transition system is explored out of a concise system description. For virtually all properties there are cases where validity can be determined long before the transition system is explored completely. For reachability, once a state satisfies $\neg\phi$ is found we can immediately return false for the query. If $\mathbf{AG}\phi$ does hold, this algorithm returns only after having explored the complete state space (the return value "true"is not explictly shown in the algorithm—assume that the algorithm returns true iff the top level call of Reachability terminates and no exit statement has been executed at any recursion level). That is, the expected space (and time) necessary to evaluate a property depends on the value of that property. Reduction techniques amplify this phenomenon, so that it is possible to verify systems not satisfying $\mathbf{AG}\phi$ which are significantly larger than systems for which $\mathbf{AG}\phi$ is true. If $\mathbf{AG}\phi$ does not hold, it can happen that the reachability algorithm terminates even if the state space of the considered system is infinite.

If we are interested in witness paths, we can output the path that is encoded in the search stack at the moment where we exited with false. When we use breadth first instead of depth first, we have to take care that we link each state to its predecessor in the search tree (for incremental depth first search, we can again use the stack).

4.2 Home properties

Home properties $\mathbf{AGEF}\phi$ (for a boolean combination ϕ of propositions), related properties of the form $\mathbf{EFAG}\phi$, as well as less frequently used formulas of the form $\mathbf{AGEFAG}\phi$ and $\mathbf{EFAGEF}\phi$[1] are closely related to *terminal* strongly connected components (tscc) of finite transition systems (with or without initial states). Throughout this section, we consider only finite state systems.

A strongly connected component C is terminal iff all successors of its elements are contained in C (i.e. no other component is reachable from C).

[1] with the CTL tautologies $\mathbf{AGEFAGEF}\phi \iff \mathbf{AGEF}\phi$ and $\mathbf{EFAGEFAG}\phi \iff \mathbf{EFAG}\phi$ for *arbitrary* ϕ, we cover all formulas that consist of a boolean combination ϕ of propositions and an arbitrarily long alternating sequence of **AG**'s and **EF**'s

Theorem 1 *$TS \models$ **AGEF**ϕ if and only if all tscc (reachable from an initial state) contain a state satisfying ϕ. $TS \models$ **EFAG**ϕ if and only if there is a tscc (reachable from an initial state) where all elements satisfy ϕ. $TS \models$ **AGEFAG**ϕ if and only all tscc (reachable from an initial state) and all their elements satisfy ϕ. $TS \models$ **EFAGEF**ϕ if and only if there is a tscc (reachable from an initial state) that contains a state satisfying ϕ.*

The proof is a simple exercise once being aware that for every reachable state there is a tscc reachable, and inside a tscc all members (and only the members) are mutually reachable.

If we want to investigate strongly connected components, breadth first search or distributed search cannot be used for the verification of home properties. For depth first search, we discuss two issues. First, we show that it is possible to simplify scc detection since it is only *terminal scc*s we are interested in. Second, we show that satisfaction of ϕ for one or for all states of a tscc can be verified on-the-fly, i.e. during the same depth first search that has to detect the tscc.

For the detection of tscc, observe that *lowlink* was defined as the smallest *dfs* of a state that is reachable via an arbitrarily long sequence of tree edges, followed by at most one edge of another kind *within the same component*. To find out whether or not an edge leads to another component, we need the data structure T in Tarjan's algorithm. Now, for tscc we know that *every* edge leads to an element of the same component. Define a new value *tlowlink* as the smallest *dfs* number of a state that can be reached via an arbitrarily long sequence of tree edges, followed by at most one edge of another kind. Compared with *lowlink*, only the additional requirement that the last state falls into the same component is removed in *tlowlink*. The number *tlowlink* can be implemented easier than *lowlink* since in minimum computations in Tarjan's algorithm, we do not need to distinguish between state in T and states outside T (lines 13-15 of Tarjan's algorithm). Then, T is no longer needed at all.

For all members of tscc, we have ($lowlink = tlowlink$) while $lowlink \geq tlowlink$ for members of nonterminal scc (the new definition can at most lead to smaller values). Thus, for all root elements of tscc, we still have $lowlink = dfs$ while for root values of nonterminal scc, this equation may or may not hold. For all states that are not root of a scc, we have $dfs > lowlink \geq tlowlink$ which means that equality between *lowlink* and *dfs* can never hold for states that are not root of a scc. That is, $tlowlink = dfs$

holds for all root states of tscc, some root states of nonterminal scc, and no other state. Hence, the remaining problem is to distinguish root states of terminal from root states of nonterminal scc. For this purpose, observe that, for every state, at least one tscc is reachable. Consequently, a root state of a nonterminal scc C has always a smaller dfs number than the root of any tscc reachable from C. Furthermore, depth first search backtracks from the tscc root (and thus detects the tscc) earlier than it backtracks from (and detects) the root of the nonterminal one. Thus, if a state is the root of a nonterminal scc then its dfs number is smaller than the dfs number of some previously detected tscc. The other way round, the dfs number of a tscc root state is always greater than the dfs number of any previously encountered *t*scc. The reason is that the root state of a tscc gets a dfs number larger than any previously visited state, and between entering this root state and backtracking from it, depth first search cannot explore any state outside that tscc (by definition of tscc). In particular, it cannot explore any other tscc, let alone assigning a larger dfs to that tscc's root. Thus, regardless of whether we use *lowlink* or *tlowlink*, the following criterion distinguishes tscc from nonterminal scc:

Theorem 2 *s is root of a tscc if and only if $dfs(s) = tlowlink(s)$ and $dfs(s)$ is greater than the dfs of the roots of all previously detected tscc.*

Proof. By definition of lowlink and tlowlink, we have for all s, $tlowlink(s) \leq lowlink(s)$. Thus, for all nodes that are not root of any scc, we have $tlowlink(s) < dfs(s)$. Furthermore, it holds $tlowlink(s) = lowlink(s)$ for all s that are members of tscc. Thus, every root of a tscc satisfies $dfs(s) = tlowlink(s)$. Since between entering and completing a tscc, no other scc is entered, the dfs of a root of a tscc is greter than the dfs of all members of previously detected tscc, including their roots.

Let s be the root of a nonterminal scc. We distinguish two cases, and show that, in case 1, $dfs(s)$ is smaller than the dfs of a previously detected scc while, in case 2, we show that $tlowlink(s) < dfs(s)$.

Case 1: There is a path from s to some tscc that contains only tree edges.

This path does necessarily contain the root s^* of this tscc. As a tree successor of s, $dfs(s^*) > dfs(s)$. Since the tree edges reflect the execution order of depth first search, the tscc of s^* has been detected before s has been completed. Thus, $dfs(s)$ is smaller than the dfs of some previously detected tscc.

Case 2: Every path from s to ant tscc contains edges other than tree edges.

Since s is root of a nonterminal scc, there is at least one path to some vertice of a terminal scc. By the case assumptions, this paths contains vertices s_1 and s_2 where $[s_1, s_2]$ is an edge, s_1 can be reached from s using only tree edges, and s_2 cannot be reached from s using only tree edges. Consequently, $[s_1, s_2]$ itself is not a tree edge. It is not a forward edge either, since a forward edge be be replaced by a sequence of tree edges. Assume, $[s_1, s_2]$ is a back edge. Then, since (by definition of back edges) s_2 must be a tree ancestor of s_1, but (by choice of s_1 and s_2) cannot be a tree descendant of s. s_2 must be a tree ancestor of s. Thus, there is a cycle starting from s, to s_1 (via tree edges), to s_2 (via $[s_1, s_2]$) back to s using tree edges. This contracdicts the assumption that s is root of an scc.

Consequently, $[s_1, s_2]$ is a cross edge. Thus, s_2 has been completed before entering s, so we have $dfs(s_2) < dfs(s)$. Since s_1 can be reached from s via tree edges, definition of tlowlink guarantees $tlowlink(s) \leq dfs(s_2)$. Thus, $tlowlink(s) \leq dfs(s_2) < dfs(s)$. $\diamondsuit$.

This criterion can be implemented by a single global variable that stores the dfs number of the root of the last recently detected tscc.

For the second task, i.e. the check whether a tscc contains states that satisfy ϕ (or do not satisfy ϕ), observe that all states of a tscc have dfs numbers greater or equal to the root's dfs number, and that no states outside a tscc can be encountered between entering and leaving a tscc during depth first search. Consider a global variable that contains the largest dfs number of a state encountered so far that satisfies ϕ (does not satisfy ϕ, resp.). Then, at the moment where we detect the tscc (are about to backtrack from its root), this number is greater or equal to the tscc root's dfs number if and only if a member of the tscc satisfies ϕ (does not satisfy ϕ, resp.).

By the above theorem this test is sufficient to verify home properties

and related properties, and does not involve a second traversal of a detected tscc. Furthermore, our new value *tlowlink* can be implemented by simply removing the condition in line 13 (and 15) from Tarjan's algorithm. With these changes, we do not need to have T as (part of a) data structure.

Fig. 4.2 summarizes the changes. We depict the **AGEF** version—algorithms for the other mentioned classes of properties look similar.

Home properties exhibit on-the-fly behavior as well. As soon as a terminal strongly connected component is detected where no state satisfies ϕ, the algorithm terminates immediately and returns false. If the home property holds, it computes the full state space.

We would like to mention that there is an alternative method to evaluate home properties. It is based on a result in [Val91b]. There, a reduction technique is proposed that guarantees that the reduced transition system contains at least one member of each tscc of the original transition system. Other than this, only few properties are preserved. Using this technique, we can, however, find a small set of states that includes at least one state of each tscc of the original transition system. For each of these nodes, we can now perform the simpler reachability verification. This approach has the advantage that the cited reduction technique, as well as reachability preserving reduction techniques may yield better reduction then a general reduction technique preserving home properties. Furthermore, we get the freedom to parallelize home property verification on a very coarse level, (the reachability problems can be solved independently of each other). We have the option to use breadth first search, or even symbolic state space verification for solving at least the reachability subproblems (as proposed in [Sch96a]).

4.3 CTL model checking

As mentioned earlier, we can view CTL formulas and all subformulas as state formulas. Thus, a CTL formula can be verified by verifying all subformulas first, for all states, and then treating these subformulas as atomic propositions for the verification of the top operator. This observation suggests a recursive verification algorithm, with a recursion depth that corresponds to the depth of the formula's syntax tree. For each state s, and each subformula ϕ of the formula to be verified, we assume a variable $l(s, \phi)$ that can take values from $\{\mathbf{true}, \mathbf{false}, \perp\}$. $\perp$ identifies that the value of ϕ in s has not yet been determined.

Figure 4.2: Depth first search for **AGEF**ϕ; exists with false iff property is false

```
1       var S: set of States initial ∅;
2       var MaxDfs: N initial 1;
3       var MaxPhi, MaxTSCC: N initial 0;
4       var dfs, tlowlink: S → N initial ∅;
5       procedure Home(current: State)
6       var new : State;
7       begin
8           dfs[current] := tlowlink[current] := MaxDfs := MaxDfs + 1;
9           if current ⊨ φ then
10              MaxPhi := dfs[current];
11          fi
12          S := S ∪ {current};
13          for new in successors(current) do
14              if new ∈ S then
15                  tlowlink[current] := MIN(tlowlink[current],dfs[new]);
16              else
17                  (update φ); Home(new); (update φ);
18                  tlowlink[current] := MIN(tlowlink[current],tlowlink[new]);
19              fi
20          done
21          if tlowlink[current] = dfs[current] and dfs[current] > MaxTSCC
                        and dfs[current] > MaxPhi then
22              exit false;
23          fi
24      end.
```

The first algorithm presented for CTL verification [CES86a] used the backward reachability relation to evaluate formulas. For instance, for a formula $\mathbf{EF}\phi$, it would start in states satisfying ϕ and set, for all states s that are backwards reachable, $l(s, \mathbf{EF}\phi)$ to **true**. This algorithm requires a preprocessed transition system with an explicitly represented (backward) edge relation.

The algorithm presented in [VL93] incorporates the evaluation of a formula on-the-fly into depth first search. Thus, we shall use this algorithm for our considerations.

The core procedures of the algorithm in [VL93] are searchAU(s, ϕ, ψ) and searchEU(s, ϕ, ψ). Their purpose is to evaluate $s \models \mathbf{A}(\phi \mathbf{U} \psi)$ and $s \models \mathbf{E}(\phi \mathbf{U} \psi)$. We may assume for simplicity that the values of ϕ and ψ are known—otherwise the verification procedure would be called recursively.

Other CTL operators can be traced back to the until operators using the CTL tautologies $\mathbf{EF}\phi \iff \mathbf{E}(\mathbf{true}\mathbf{U}\phi)$, $\mathbf{AF}\phi \iff \mathbf{A}(\mathbf{true}\mathbf{U}\phi)$, $\mathbf{AG}\phi \iff \neg\mathbf{EF}\neg\phi$, $\mathbf{EG}\phi \iff \neg\mathbf{AF}\neg\phi$. For the next step operators **AX** and **EX**, investigation of the immediate successors is sufficient and can be implemented easily.

Procedures searchEU and searchAU each perform a single depth first search. The goal of searchEU is to find a witness path, i.e. a sequence of successive states where the last element satisfies ψ and all other states satisfy ϕ. The goal of searchAU is to find a counterexample path, i.e. either a path of successive states where the last element satisfies neither ϕ nor ψ while the other ones do not satisfy ψ, or an infinite sequence of states satisfying ϕ but not ψ (for a finite state system, if there is an infinite sequence, there is also a sequence ending in a cycle[2]). These two are exactly the possibilities that can make a universal until formula false, as expressed in the following CTL tautology:

$$\neg\mathbf{A}(\phi \mathbf{U} \psi) \iff \mathbf{EG}(\phi \wedge \neg\psi) \vee \mathbf{E}(\neg\psi \mathbf{U}(\neg\phi \wedge \neg\psi))$$

.

As soon as the goal is achieved, or could not be achieved after having investigated the whole search space, the procedure stops searching and returns. This is another example of the on-the-fly principle. The search space,

[2]Consider a state without enabled actions as a special case of a cycle—remind Def. 5 of *path* that assumes terminal states to be repeated indefinitely in infinite paths.

for both procedures, consists of states s with the following properties (and their immediate neighbors in the transition system):

- $l(s, \phi) = \mathbf{true}$ (or $\perp$, and recursive verification evaluates it to **true**);
- $l(s, \psi) = \mathbf{false}$ (or $\perp$, and recursive verification evaluates it to **false**);
- $l(s, \mathbf{A}(\phi \mathbf{U} \psi)) = \perp$ (for searchAU), or $l(s, \mathbf{E}(\phi \mathbf{U} \psi)) = \perp$ (for searchEU);

If, at the end of a (possibly empty) sequence of states satisfying the mentioned conditions, searchEU hits a state where ψ is true, we have the witness path we were looking for and can terminate search. The same is true if we hit a state where $\mathbf{E}(\phi \mathbf{U} \psi)$ has already been evaluated to true, say, by a previous run of searchEU. In that case, the current path, extended by the witness path from the previous run of searchEU, is a valid witness path for the current run of searchEU. If searchEU hits a state where both ψ and ϕ are false, no extension of the path can ever become a witness path for the existential until operator, so we do not need to search beyond that state. The same is true if $\mathbf{E}(\phi \mathbf{U} \psi)$ is already known to be false in some state.

Dually, if searchAU hits a state satisfying ψ or known to satisfy $\mathbf{A}(\phi \mathbf{U} \psi)$, the current search path can never be extended to a counter example, since it is guaranteed to satisfy $\phi \mathbf{U} \psi$. If we hit a state where neither ϕ nor ψ hold, or $\mathbf{A}(\phi \mathbf{U} \psi)$ is already known to be false, we have the counterexample we are looking for (or can extend the current search path with the counterexample from the previous run).

Following the description so far, the model checking procedure appears to have a worst case time complexity $O(|\phi| \cdot (|S| + |E|)^2)$ where $|\phi|$ is the length of the CTL formula, and $[S, E, (A)]$ the transition system—for every state, and every subformula of ϕ, we launch at most once a search procedure of $O(|S| + |E|)$ complexity. The algorithm in [VL93], as well as the algorithm in [CES86b] require, however, only $O(|\phi| \cdot (|S| + |E|))$. The reason, in all cases, is that different calls of searchEU and searchAU do not overlap, i.e. they are performed on disjoint subsets of S. For the [VL93] algorithm, this is achieved by an additional capability of searchEU and searchAU. After each call of searchEU(s, ϕ, ψ) (as well as searchAU(s, ϕ, ψ)), not only s will have received its value for the until formula, but also all states that have been visited during that particular search. These states do not require another run of searchEU/searchAU on ϕ and ψ.

Consider a run of searchAU that leads to $s \models \mathbf{A}(\phi \mathbf{U} \psi)$. In that case, no node that has been visited during search can violate the until formula since otherwise, as already discussed, the path from s to that node extended by that node's counterexample would be a counterexample for s as well. The other way round, assume that search stops at a counterexample path (i.e. a path to a state satisfying neither ϕ nor ψ, or a sequence of states ending in a loop, identified by a back edge into the search stack). Every element on the counterexample path, i.e. every state that is on the stack in the moment the counterexample has been found, does not satisfy $\mathbf{A}(\phi \mathbf{U} \psi)$ since the suffix of the counterexample starting in that state is a counterexample path itself. States that have been visited during search but are no longer on the search stack satisfy $\mathbf{A}(\phi \mathbf{U} \psi)$. Call this set of states N, and the set of states on the stack T. These nodes were on the stack earlier, and no counterexample had been found at that time. That is, there is no cycle and no state satisfying neither ϕ nor ϕ reachable only using states in N. Any path involving other states would contain states of T since depth first search backtracks from a state only after having explored all its successors. The following consideration shows that states in N cannot have successors in the stack T. Let $n \in N$. n must have ancestors in the search tree that are in T (at least, the initial state is in T). Let t be the ancestor of n in T with largest dfs number and assume that there is a path from n to some element t' of T. If the dfs number of t' is smaller or equal to the dfs number of t then n, t, and t' are on a cycle. In that case, the back edge to t' would have occurred while n was on the search stack which did not happen (since otherwise that would have identified a counterexample cycle and search would have stopped there). On the other hand, if the dfs number of t' is larger than the dfs number of t then t' would have been explored by depth first search from n and been backtracked from earlier than n which contradicts the assumption that t' is in T. Thus, nodes in N cannot have successors in T and states outside $N \cup T$, nor is there a counterexample only involving elements of N. Consequently, states in N satisfy $\mathbf{A}(\phi \mathbf{U} \psi)$. This value can be set at the moment states are removed from the stack.

Consider now a run of searchEU that leads to $s \not\models \mathbf{E}(\phi \mathbf{U} \psi)$. We have $s' \not\models \mathbf{E}(\phi \mathbf{U} \psi)$ for all states s' covered by that search since otherwise the path from s to s', extended by the witness path at s' yields a witness path for s. For the case that we have $s \models \mathbf{E}(\phi \mathbf{U} \psi)$, we have to distinguish three kinds of nodes. The first is the set of states on the stack in the moment of finding the witness path. These states satisfy $\mathbf{E}(\phi \mathbf{U} \psi)$ since the suffix of the

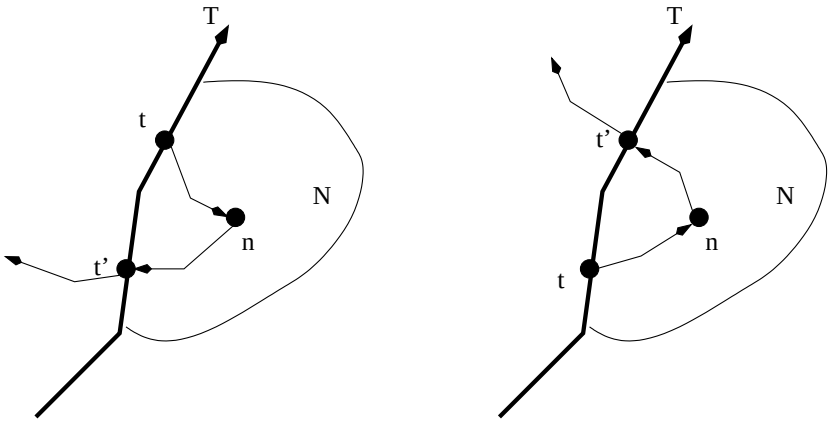

Figure 4.3: Two situations in searchAU where a state n has been visited, but is no longer on the stack. t is closest ancestor of n on the current stack (depicted as thick arrow), t' is the state on the stack it would hit under the assumption that there is a path from n out of the set N of states that have left the stack.

witness path starting at them is a witness path. States that are no longer on the stack but share a scc with members of the stack satisfy $\mathbf{E}(\phi\mathbf{U}\psi)$ as well since they can reach a member of the stack and connect to that state's witness path (remember that all elements in the search space satisfy ϕ). The third kind of node is those that are no longer on the stack and do not share a scc with nodes on the stack. These nodes do not satisfy $\mathbf{E}(\phi\mathbf{U}\psi)$ since a scc is completed only after all reachable scc have been completed, too, obviously without having found a witness path. The three kinds of nodes can be easily distinguished using Tarjan's algorithm. Nodes on the stack and nodes sharing a scc with nodes on the stack receive the same value—they are exactly the elements of the data structure T. Elements of completed scc are those elements of S that have been removed from T.

These considerations lead to the following implementation of explicit state CTL verification. We present the algorithm of [VL93] in Tarjan terminology while in [VL93] they use some equivalent to *lowlink* for detecting scc.

For searchAU, notice that setting l(current,$\mathbf{A}(\phi\mathbf{U}\psi)$) to true already in line 8 permits a uniform treatment of cycle detection with the other ways of finding a counterexample.

Figure 4.4: Depth first evaluation of universal until formulas

```
1   var S: set of States initial ∅;
2   var l: S× subformulas → {true, false, ⊥} initial ⊥;
3   var found: boolean initial false;
4   procedure searchAU(current: State,φ,ψ:formula)
5   var new : State;
6   begin
7       if l(current,A(φUψ)) = true or CTL(current,ψ) then
8           l(current,A(φUψ)) := true; return;
9       fi
10      if l(current,A(φUψ)) = false or not CTL(current,φ) then
11          l(current,A(φUψ)) := false;
12          found := true; return;
13      fi;
14      l(current,A(φUψ)) := false;
15      S := S ∪ {current};
16      for new in successors(current) do
17          if new ∉ V then
18              searchAU(new,φ,ψ);
19              if found then
20                  return;
21              fi
22          fi
23      done
24      l(current,A(φUψ)) := true;
25  end.
```

Figure 4.5: Depth first evaluation of existential until formulas

```
1  var S,T: set of States initial ∅;
2  var l: S× subformulas → {true, false, ⊥} initial ⊥;
3  var found: boolean initial false;
4  var MaxDfs: N initial 0;
5  var dfs, lowlink: S → N initial ∅;
6  procedure searchEU(current: State,φ,ψ:formula)
7  var new : State;
8  begin
9     if l(current,E(φUψ)) = true or CTL(current,ψ) then
10       l(current,E(φUψ)) := true;
11       for all s ∈ T do
12          l(current,E(φUψ)) := true;
13       done
14       found := true; return;
15    fi
16    if l(current,E(φUψ)) = false or not CTL(current,φ) then
17       l(current,E(φUψ)) := false; return;
18    fi
19    l(current,E(φUψ)) := false;
20    dfs[current] := lowlink[current] := MaxDfs := MaxDfs + 1;
21    S := S ∪ {current}; T := T ∪ {current};
22    for new in successors(current) do
23       if new ∈ S then
24          if new ∈ T then
25             lowlink[current] := MIN(lowlink[current],dfs[new]);
26          fi
27       else
28          searchEU(new,φ,ψ);
29          if found then
30             return;
31          fi
32          lowlink[current] := MIN(lowlink[current],lowlink[new]);
33       fi
34    done
35    if lowlink[current] = dfs[current] then
36       T := T \ {v | v ∈ T ∧ dfs[v] ≥ dfs[current]};
37    fi
38 end.
```

Figure 4.6: Verification of CTL formulas

```
1   procedure CTL(s : state, φ: formula) : boolean;
2   begin
3       if l(s,φ) ≠ ⊥ then
4           return l(s,φ);
5       fi
6       case φ is
7       proposition:    l(s,φ) := EvaluateProposition(s,φ);
8       φ1 ∧ φ2:        l(s,φ) := CTL(s,φ1) and CTL(s,φ2);
9       φ1 ∨ φ2:        l(s,φ) := CTL(s,φ1) or CTL(s,φ2);
10      ¬φ1:            l(s,φ) := not CTL(s,φ1);
11      E(φ1Uφ2):       searchEU(s,φ1,φ2);
12      EFφ1:           searchEU(s,true,φ1);
13      AGφ1:           searchEU(s,true,¬φ1);l(s,φ) := not l(s,E(trueUφ1));
14      A(φ1Uφ2):       searchAU(s,φ1,φ2);
15      AFφ1:           searchAU(s,true,φ1);
16      EGφ1:           searchAU(s,true,¬φ1);l(s,φ) := not l(s,A(trueUφ1));
17      EXφ1:           l(s,φ) := false;
18                      for new in successors(s) do
19                          if CTL(new,φ1) then
20                              l(s,φ) := true;
21                          fi;
22                      done;
23      AXφ1:           l(s,φ) := true;
24                      for new in successors(s) do
25                          if not CTL(new,φ1) then
26                              l(s,φ) := false;
27                          fi;
28                      done;
29      esac;
30      return l(s,φ);
31  end.
```

4.4 LTL model checking

An LTL formula can be seen as a description of a set of infinite paths (an ω-language). As alphabet, we use assignments of truth values to the atomic propositions occurring in the formula, i.e. boolean vectors. In [VW86], it was shown that every ω-language defined by an LTL formula is ω-regular, i.e. is accepted by a finite Büchi-automaton. A Büchi-automaton is an extension of the concept of automata with acceptance states to infinite words. The main characteristics is that an infinite word is accepted by a Büchi-automaton if and only if the automaton, executed on that word, enters states contained in the acceptance set infinitely often. The concept has been extended to several acceptance sets, now requiring that each acceptance set has to be visited infinitely often. Given an LTL formula, a corresponding Büchi-automaton can have, in worst case, a number of states that is exponential in the length of a formula. Algorithms used today for constructing Büchi-automata out of an LTL formula, however, tend to produce reasonably small automata [GPMW95].

A transition system defines a Büchi-automaton as well (treating all states as acceptance states). If we label states of the transition system by the boolean vector of values of atomic propositions that are satisfied in that state, we have two Büchi-automata operating on the same alphabet. The LTL model checking problem reduces to the question whether the ω-language defined by the transition system is included in the language defined by the formula. This language inclusion problem is solved indirectly, by asking instead whether the intersection of the language defined by the transition system and the complement of the language defined by the formula is empty. Complementing the formula's language is done by constructing an automaton for the negated formula rather than for the formula itself. Given two Büchi-automata, a third Büchi-automaton can be constructed that accepts the intersection of the languages defined by the two given automata. The set of states of that automaton is the cross-product of the state sets of the two original automata, so the constructed automaton has as many states as the product of the sizes of the given automata. On the product automaton, emptiness of the accepted language can be checked by depth first search for a cycle in the automaton that visits all acceptance sets. If such a cycle exists, the path from the initial state to that cycle, plus infinite iteration through that cycle, correspond to a counterexample execution of the transition system. If a counterexample exists in a finite state product automaton, it must

visit some states infinitely often, so there must be a cycle as well.

In this thesis, we are not going to consider LTL model checking in depth. For our purposes, it is sufficient to understand that LTL model checking corresponds to depth first search on the product automaton, i.e. a structure that is as many times as large as the transition system, as there are states in the Büchi-automaton representing the negated formula to be verified.

4.5 Liveness properties

In this section, we discuss explicit state space techniques that are particularly dedicated to goal, stabilization, and immortality properties. In contrast to general LTL model checking, these techniques operate on the plain transition system, i.e. we do not build a product with any kind of automaton. Thus, the worst case space requirement for checking a property shrinks from $O(r \cdot b)$ where r is the size of the transition system, and b the number of states in the Büchi automaton representing the LTL formula, to just $O(r)$. As the properties considered here are all liveness properties, they share the need of obeying fairness requirements in their verification. Our algorithms require a run time of $O(r \cdot f)$ where r is the size of the transition system, and f is the number of *strong* fairness requirement involved. Weak fairness requirements can be checked on-the-fly and do not contribute to the asymptotic worst case complexity of our algorithms. The implementation of fairness requirements used here is due to [LP85].

Fairness is related to infinite paths of the system. Removing any finite prefix of a path does not alter validity of any fairness requirement. A fairness requirement depends therefore only on those states that occur infinitely often in a path. The set of states that occur infinitely often in a path is strongly connected. Thus, it is reasonable to rephrase fairness requirements in terms of strongly connected sets of states.

Definition 11 (Fair sets of states) *Let S be a strongly connected set of states. S satisfies a weak fairness requirement ϕ iff it contains a state satisfying ϕ.*

S satisfies a strong fairness requirement $[\phi, \psi]$ iff there is no state in S satisfying ϕ, or there are states in S satisfying ψ.

It is easy to verify that

Proposition 4 (Fair paths correspond to fair sets) *A path satisfies a set of fairness requirements if and only if the set of states occurring infinitely often in that path does.*

For every strongly connected set of states, there is a path that has exactly this set as the set of states occurring infinitely often.

Observe that we consider strongly connected *sets* which are not necessarily strongly connected *components*.

The first claim follows immediately from the definitions, the second one from strong connectedness.

In contrast to infinite paths, strongly connected sets are finite structures, so they are suitable for verification.

Verification of all three classes of properties relies on searching for counterexamples. Thereby, a counterexample is a fair strongly connected set of states.

Theorem 3 (Counterexamples of liveness properties) *Let* $[S, E, A]$ *be a transition system where all states are reachable from initial states, and assume the presence of some weak and/or strong fairness requirements. Let* ϕ *be a state predicate.* $[S, E, A]$ *satisfies*

- $\mathbf{GF}\phi$ *iff there is no strongly connected set of states in* $[S, E, A]$ *that satisfies all fairness requirements and does not contain states satisfying* ϕ*;*
- $\mathbf{FG}\phi$ *iff there is no strongly connected set of states in* $[S, E, A]$ *that satisfies all fairness requirements and contains states not satisfying* ϕ*;*
- $\mathbf{F}\phi$ *iff there is no strongly connected set of states in* $[S, E, A] \setminus \phi$ *that satisfies all fairness requirements.*

Thereby, $[S, E, A] \setminus \phi$ *is the restriction of* $[S, E, A]$ *to all states that can be reached from initial states without passing states that satisfy* ϕ*.*

Proof. If there is a fair path not satisfying $\mathbf{GF}\phi$, this path must satisfy $\mathbf{FG}\neg\phi$. That is, all states occurring infinitely often in this path do not satisfy ϕ and therefore, there is a fair strongly connected set of states not satisfying ϕ. The other way round, given a fair strongly connected set of states not satisfying ϕ, the above proposition states the existence of a fair

path where none of the infinitely often occurring states satisfies ϕ. Thus, this path violates $\mathbf{GF}\phi$.

The claim for $\mathbf{FG}\phi$ can be proven similarly.

For $\mathbf{F}\phi$ observe additionally, that every fair path violating $\mathbf{F}\phi$ corresponds to a fair strongly connected set in $[S, E, A] \setminus \phi$, and every fair strongly connected set in $[S, E, A]\setminus\phi$ can be extended to a fair path of states not satisfying ϕ, i.e. a counterexample for $\mathbf{F}\phi$. $\diamondsuit$

The core of our search for fair strongly connected counterexamples is a procedure that checks, given a set S^* of states, whether this set contains fair strongly connected subsets. This procedure is based on the following observations.

First, every strongly connected set in S^* is included in some strongly connected component of S^*. Thus, we start by computing strongly connected components of S^*. This takes $O(|S^*| \cdot |E^*|)$ time where E^* is the set of edges in the original transition system that connects states in S^*. Each component can be checked whether the whole component itself satisfies all weak and strong fairness constraints. This is easily done by a single traversal of the component. If all fairness constraints are satisfied for such a component, this component forms a fair strongly connected set. If a weak fairness constraint is violated, all strongly connected subsets violate that constraint as well. So, we do not need to investigate any subset of that component. A component violating a strong fairness constraint $[\phi, \psi]$ means that it contains states satisfying ϕ but no states satisfying ψ. Since none of the subsets of the component can contain states satisfying ψ, only subsets not containing states that satisfy ϕ can satisfy the fairness requirement. Thus, we proceed to search for fair strongly connected sets with the set of states that is obtained through removing all ϕ-states from the investigated component. All subsets of the newly formed set of states do obviously satisfy the requirement $[\phi, \psi]$. Thus, depth of recursion cannot exceed the number of specified strong fairness requirements.

This basic procedure from [LP85] can be adapted to a verification algorithm for all three classes of liveness properties. For goal properties, we apply the search procedure to all strongly connected components of $[S, E, A] \setminus \phi$. For immortality properties, we apply it to all sets of states that are formed by removing all ϕ–states from strongly connected components of $[S, E, A]$. For stabilization properties, we apply the search for fair strongly connected sets to plain strongly connected components of $[S, E, A]$ but add $\neg\phi$ as a weak fairness requirement. This way, only fair strongly connected sets containing

states that violate ϕ are produced.

Due to the similarities between these three procedures, only the core procedure of searching for fair strongly connected subsets of a given (not necessarily connected) set of states is presented in Fig. 4.7.

Figure 4.7: Search for fair strongly connected subsets of S^* in a transition system $[S, E, A]$

```
1   const WF: set of state predicates; /* weak fairness requirements */
2   const SF: set of pairs of state predicates; /* strong fairness req. */
3   var found: bool initial false;
4   function ContainsFairSet(S*: set of states) : bool
5   begin
6     for all C : C is strongly connected component of [S*, E, A] do
7       for all φ ∈ WF do
8         if {s | s ∈ C and s ⊨ φ} = ∅ then
9           goto next;
10        fi
11      done
12      for all [φ, ψ] ∈ SF do
13        if {s | s ∈ C and s ⊨ ψ} = ∅ and {s | s ∈ C and s ⊨ φ} ≠ ∅}
            then
14          if ContainsFairSet(C \ {s | s ⊨ φ}) then
15            return true;
16          else
17            goto next;
18          fi
19        fi
20      done
21      return false;
22  next:
23    done
24  end.
```

Part III

Reduction techniques

This part is concerned with four classes of techniques for reducing the size of a state space to be explored. For each of these classes, presentation consists of five sections. In the first sections, we study the theory that underlies the reduction technique. It contain the arguments as to why the methods preserve the properties they are supposed to preserve, and why certain computations produce the desired results. In the second sections, we present implementations and study efficiency issues. A third section illustrates the performance of the method using our running examples and, if necessary, a few additional examples. In a dedicated section on compatibility, we discuss possibilities to apply the studied method with other methods, introduced earlier. We do also discuss restrictions concerning the choice of search strategy applied for a reduction technique. The fifth section summarizes the method.

Chapter 5

Stubborn sets

The stubborn set method [Val88, Val91b, Val98] is an instance of a class of techniques frequently called *the partial order reduction*. Other instances of partial order reduction are the *persistence set* method [GW91] and the ample set technique [Pel93]. All partial order reduction methods are based on the observation that, particularly in distributed systems, many permutations of an executable sequence of actions are executable as well and lead to the same final state. Thereby, the permutations create a huge number of different intermediate states which contribute significantly to state space explosion. In distributed systems, a large number of executable permutations of a sequence of actions occurs typically in situations where several processes execute local actions, see Fig. 5.1.

Applying partial order reduction means selecting a subset of the set of enabled actions in each state and exploring only the selected actions, while the remaining enabled actions in that state are discarded. This way, a subsystem (an underapproximation) of the original transition system is constructed. Paths in the original transition system that contain discarded actions are not present in the subsystem, and many intermediate states become unreachable.

The loss of discarded paths is compensated by designing the selection scheme in a way that every discarded path corresponds to a preserved path that is indistinguishable with respect to the property to be verified, so the subsystem holds some given property if and only if the original transition system does.

The earliest stubborn set method [Val88] was a technique that preserved all deadlock states of a system. Furthermore, the reduced system contains an infinite execution if and only if the original system does. Most subsequent

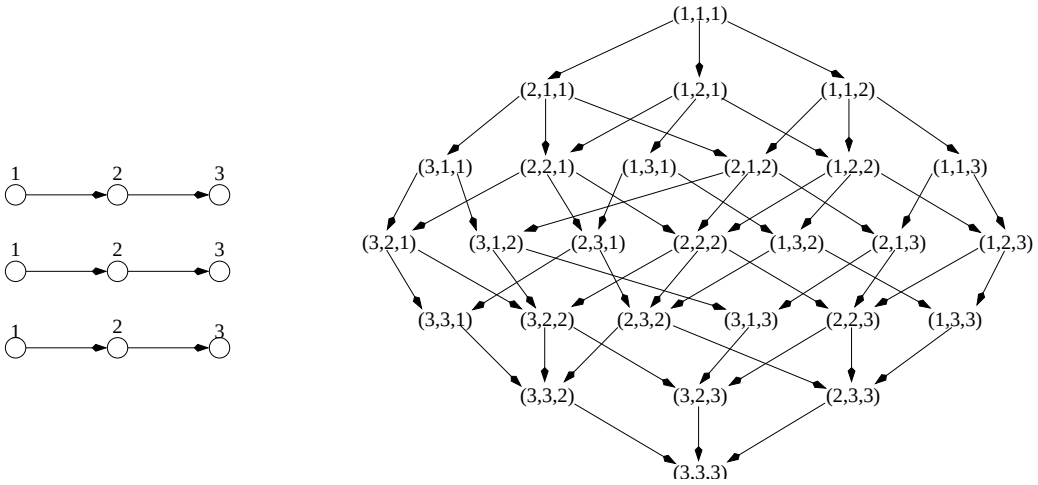

Figure 5.1: Three independent (parts of) processes consisting of three states each (left) span a huge state space when they can execute concurrently inside a distributed system.

developments refined the capability of that technique to preserve at least one infinite execution. Using a more elaborated selection scheme, it was possible to better control *which* kind of paths one would like to preserve in the subsystem. So, for instance, [Val91b] proposed a selection scheme that preserved a large class of safety properties (by guaranteeing that every counterexample path in the original system corresponds to a counterexample path preserved in the subsystem). In this method, the concept of *visible* and *invisible* actions was introduced. An action is invisible iff its occurrence does not influence the validity of elementary propositions occurring in a temporal logic formula to be verified. Preservation of a property was proven by showing that each finite sequence of actions in the original transition system corresponds to a sequence in the subsystem where the same visible actions occur in the same order, but interleaved with possibly different invisible actions, not necessarily in the same order. Though the notion of visibility can be relaxed in some situations [KPV97, Var98], it recurs in many existing stubborn set methods. Hence, stubborn sets based on invisibility rely on the existence of a large number of invisible actions since otherwise too many paths (and states, too) would be preserved. So, the technique yields excellent results first and foremost for local properties. An alternative concept for safety properties [Val91b, Val93], that codes a property by some modifications in the system

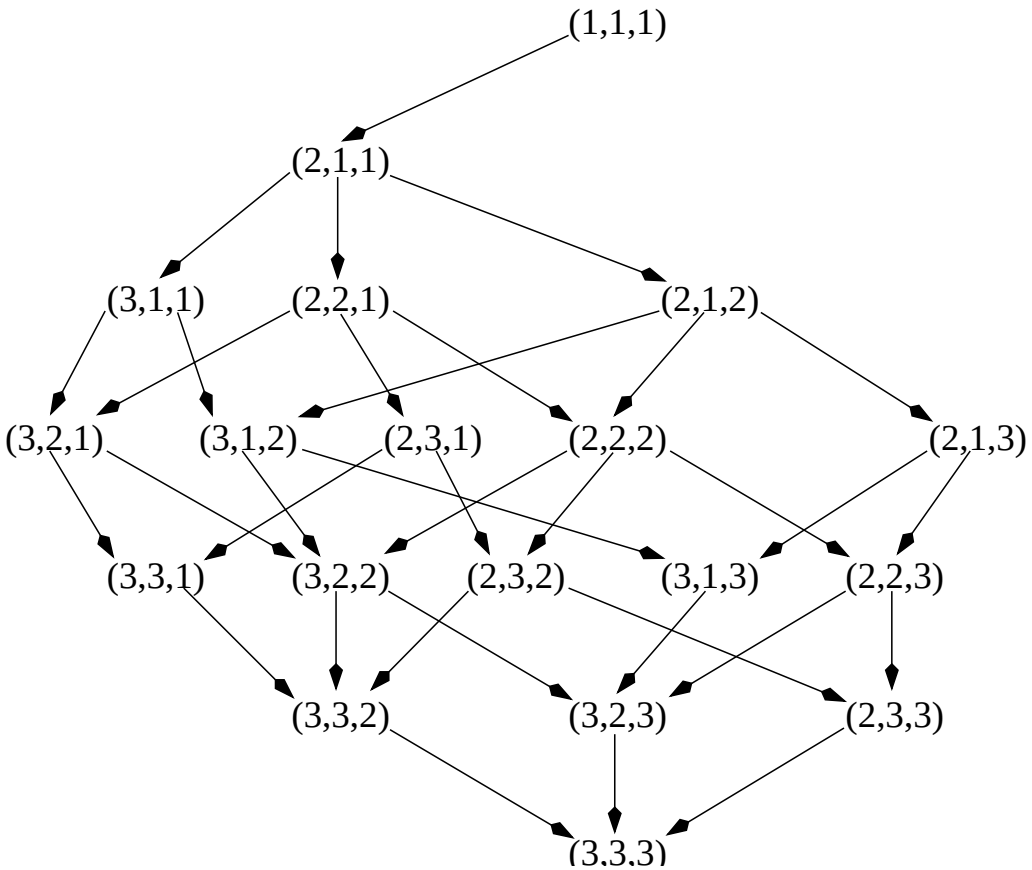

Figure 5.2: If only the event of the first component is considered in state (1,1,1) of the above system, the state space shrinks. In addition to the two immediate successors, 6 more states become unreachable.

to be verified, does not rely on visible and invisible actions, but leads to similar conclusions about locality of properties. This is due to the fact that a process is added to the original system that "monitors"the property to be verified. For a global property, there tend to be more links to the original system than for a local property. Tightly interlinked systems, however, tend to yield inferior results in the stubborn set reduction. The concept of visibility, and therefore the restriction to local properties, retains in partial order reduction techniques for larger classes of properties, including the LTL preserving methods [Val92, GW91, Val93, Pel93] and the CTL*-preserving partial order reduction method [GKPP95]. The surveys [Val96, Val98] contain large lists

of stubborn set like approaches to various classes of properties.

In [Sch99b], we proposed stubborn set methods that are capable of preserving several classes of Petri net state properties. Among the preserved properties are several global properties, that is, properties where few or no actions can be treated as invisible. In fact, our method can be seen as an immediate successor of the capability of [Val88] to preserve deadlock states. Thus, the path oriented partial order reduction branch leading from [Val88] via the safety-property and LTL preserving methods to the CTL*-preserving partial order reduction technique, has been complemented by a state oriented branch. The new state oriented branch reaches into parts of CTL [Sch00e]. In [KV00], the new state oriented stubborn set methods were significantly improved by relaxing the conditions on the action selection scheme proposed in [Sch99b].

5.1 Theory

This section is concerned with sufficient criteria under which a selection scheme for generating a reduced transition system preserves a given property. The next section deals with algorithms that compute a set of selected actions, only relying on information available while state space generation is in progress. In both sections, we focus on state oriented stubborn sets. Path oriented stubborn sets are surveyed only briefly and incompletely—we present them mainly for comparison purposes. Furthermore we restrict ourselves to the case of deterministic actions. Most approaches can be extended to non-deterministic actions, but the deterministic case involves conditions that are simpler and easier to read. For more details on path oriented stubborn set methods, we refer the reader to the associated publications mentioned in the previous section.

The main concept in this section is that of a *stubborn set*. A stubborn set is a set of actions associated to a state. The enabled actions inside a stubborn set are those selected for exploration in the reduced transition system, the enabled actions outside the stubborn set are the discarded ones. The stubborn set itself may contain both enabled and disabled actions. An assignment of stubborn sets to each state defines a reduced transition system that contains all initial states of the original system, and contains, for each state s in the reduced system, exactly the successor states s' that correspond to edges $[s, a, s']$ with a being an element of the stubborn set. Throughout

this section, the transition system obtained this way—using stubborn sets determined by the respective context—is referred to as *the reduced transition system*. The set of conditions required for a stubborn set depends on the properties to be verified. However, the following concept of *weak stubbornness* is common to all versions of stubborn sets:

Definition 12 (weak stubborn set) *Let $[S, E, A]$ be a transition system. A set U of actions is* weak stubborn *in a state $s \in S$ iff, for all finite sequences w of actions outside U, every state s', and every action $a \in U$, $s \xrightarrow{wa} s'$ implies $s \xrightarrow{aw} s'$.*

Weak stubbornness reflects the ability to execute two permutations wa and aw of a sequence, one of which (wa) is discarded by the stubborn set reduction (since the first action of w is not in the stubborn set and thus not explored in reduced transition system generation), while for the other sequence (aw) at least the first action is explored. Whether or not the whole aw is explored depends on the stubborn sets in successor states of s.

Weak stubborn sets as such do not preserve any meaningful properties. For instance, the empty set of actions is weak stubborn in all states. The basic stubborn set method [Val88] adds the following requirement to weak stubborn sets.

Definition 13 (basic stubborn sets) *Let $[S, E, A]$ be a transition system. A weak stubborn set U is* basic stubborn *in a state $s \in S$ iff either there is no enabled action in s, or there is an action $a \in U$ such that for every sequence w of actions outside U, sequence wa can be executed in s.*

An action that fulfills the requirement for a is called *key action*. Def. 13 implies that a key action is enabled in s (use the empty sequence for w).

Theorem 4 ([Val88])

1. *A reduced transition system that is constructed using basic stubborn sets in every state contains all deadlock states (states without enabled actions) of the original transition system;*

2. *A reduced transition system that is constructed using basic stubborn sets in every state contains an infinite path not ending in a deadlock state if and only if the original transition system does.*

The first claim shall turn out to be a consequence of Thm. 10 later in this section. The second claim follows, by simple induction, from the observation that each state s in the reduced state space from which an infinite sequence of actions can be executed, has a successor state in the reduced state space from which an infinite sequence can be executed. To verify the observation, consider two cases.

In the first case, the infinite sequence executable at s contains an action that is member of the basic stubborn set U used at s. Then, the sequence can be written as waw' where w' is a finite sequence of actions outside U, $a \in U$, and w' is an infinite sequence of arbitrary actions. By weak stubbornness, aww' can be executed at s as well, so the infinite sequence ww' can be executed from a–successor of s, which is contained in the reduced state space.

In the second case, the infinite sequence does not contain elements of the basic stubborn set U used in s. Then it can be verified that the same infinite sequence as in s is executable in the a–successor of s where a is any of the key actions in U whose existence is required by Def. 13.

Path oriented stubborn set methods link to the second claim of Thm. 4. The idea is to view the proof of that claim as a process that, given a path in the original transition system, constructs a corresponding path in the reduced transition system. So far, these two paths have not much in common beyond their infinity. According to case one of the proof sketch, actions of the original path may appear in different order, according to case two there may be actions in the new path that have not occurred in the original path at all. Since it can happen that case two is used indefinitely, the new path can consist of totally different actions than the original path. Of course, we have to accept that a path in the reduced transition system may look different from the original path, but for preserving advanced path related properties we need to ensure that some features of the original path are passed to the corresponding path in the reduced transition system. The two concepts serving this goal are *visibility* and *ignorance*.

Consider a partition of the set A of actions into a set V of *visible* and a set $\overline{V}$ of *invisible* actions. For the visible actions, we would like to assure that the order of their occurrence in the original path is preserved in the new path of the reduced transition system while invisible actions may be arbitrarily interleaved, removed, inserted, and permuted.

Consider the requirement that a stubborn set does always contain either all visible actions or no visible action at all, and return to the case consider-

ation for the second claim of Thm. 4. In case one, the original sequence is modified by shifting the first element of the stubborn set U in the original sequence to the front of the sequence. If the shifted action is visible then all actions in front of it are invisible since they are outside U in this case, and all visible actions are inside U. So, whether or not the shifted action is visible, the relative order of visible actions in waw' is the same as in aww'. The second case applies if the original sequence does not contain elements of U at all. That is, if the original sequence contains visible actions, U consists only of invisible actions, including the key action used to continue the construction of the original path. If the original sequence does not contain visible actions then the reduced path constructed to this point contains already all visible actions of the original path, in preserved order.

With the all-or-none criterion for visible actions, we achieve that *if* visible actions of the original sequence are used in the constructed reduced sequence, their relative order of appearance remains the same. It does not, though, assure that they are ever included into the constructed path. It can still happen that the stubborn sets consist indefinitely of invisible actions other than those in the original sequence. This phenomenon is treated by the concept of *ignored actions*.

Definition 14 (ignored action) *An action a is ignored in a reduced transition system iff the reduced system has a terminal strongly connected component C where a is enabled in all $s \in C$ but not in the stubborn set used in any of the $s \in C$.*

Together with the requirement that *every* enabled action in a stubborn set be a key action, it can be shown that a reduced transition system where no action is ignored, preserves paths of the original transition in the following sense: if s is a state in the reduced transition system and $a_1 a_2 \ldots a_n$ is a sequence of actions executable in the original system, then there is a permutation π of $\{1, \ldots, n\}$ such that some path of actions $\ldots a_{\pi(1)} \ldots a_{\pi(2)} \cdot \cdots \cdot a_{\pi(n)}$ is part of the reduced transition system. This claim can be shown according to the following idea: with all enabled actions in the stubborn set being key actions, indefinite non-applicability of case 1 in the basic construction of the reduced path means that we can navigate freely through the reduced state space using case 2. Arriving in a terminal strongly connected component and traversing it completely, the first action of the infinite sequence that remains unchanged by application of case 2, turns out to be an ignored action, in contradiction to the construction. Thus, in absence of ignorance, and with all

enabled actions being key actions, every element of the original sequence can eventually be consumed in the construction of the corresponding sequence of the reduced state space.

The informal considerations on the ignorance phenomenon can be detailed out to prove the next result.

Definition 15 (Strong stubborn sets) *A basic stubborn set U is* strong stubborn *in a state s iff for all enabled $a \in U$, all states s', and all sequences w of actions outside U, $s \xrightarrow{w} s'$ implies that a is enabled in s'. In other words, all enabled actions are key actions.*

Theorem 5 ([Val91b]) *Consider a reduced transition system where no action is ignored and, in each state, strong stubborn sets are used. Then*

1. *if an action is enabled in some reachable state of the original transition system, it is enabled in some state of the reduced transition system, too;*
2. *if a transition of a Petri net is live ("**AGEF**t enabled"holds) in the original transition system, this transition is live in the reduced system, too;*
3. *for every terminal strongly connected component C of the original transition system, the reduced transition system contains at least one $s \in C$.*

Together with our earlier considerations about visible actions, we obtain

Theorem 6 ([Val91b]) *Consider a reduced transition system where no action is ignored and, in each state, strong stubborn sets are used that contain either all or no visible action. Then, for every state s in the reduced system, and finite sequence w of actions executable in s in the original transition system, there is a finite sequence w' of actions starting in s in the reduced transition system such that the projections of w and w' onto the set of visible actions coincide.*

This theorem can be used to prove preservation of stuttering insensitive temporal logic properties where satisfaction or violation relates to finite witness or counterexample paths. This is the case for various safety properties. All we need to do is to let the set of visible actions include those that potentially alter validity of the elementary propositions in the formula. For instance, in a variable based approach, a proposition $x < y$ would cause at

least all guarded commands to be visible that assign to x or y, commands that alter the expressions to be assigned, and so on. Then, for each witness or counterexample in the original transition system, there is a path in the reduced system that is equivalent on the visible actions and thus creates the same sequence of changes to elementary propositions of the formula. In a stuttering insensitive property, the new path is a valid witness or counterexample, too. Properties that are covered by this approach include reachability and home properties.

A temporal property is *stuttering insensitive* iff it does not distinguish between executions that assign the same values to elementary propositions for a different number of consecutive states in the execution. Reachability properties $\mathbf{EF}\phi$ are stuttering insensitive since it is irrelevant how many consecutive states satisfy $\neg\phi$ before a state satisfying ϕ is reached. As a matter of fact, all CTL* properties not containing the $\mathbf{X}$ operator are stuttering insensitive. Stuttering insensitivity is crucial for the above approach since the correspondence between paths in original and reduced transition systems concerns only the visible actions while their interleaving with invisible actions is in part uncorrelated. Invisible actions do not change elementary propositions of the formula but may well change the number of consecutive states assigning the same values to those propositions.

Thm. 6 does not extend to infinite sequences. For instance, if a sequence in the original transition system contains only finitely many visible actions, the construction based on the case consideration for Thm. 4 lasts only to the moment where all visible actions of the original path have been inserted in the new path by case 1. From that moment on, case 2 can be applied indefinitely with visible actions as key actions in U. That is, an infinite path where some assignment to elementary propositions holds indefinitely from some point on, may be not reflected in the reduced transition system. This may alter the value of LTL liveness properties. In order to treat LTL liveness properties, the following requirements have been established [GW91, Pel93, Val92, Val93].

Definition 16 (LTL preserving stubborn set) *Let V be a set of actions (the set of visible actions). Let U be a basic stubborn set in state s such that*

- *U contains either all or no element of V;*
- *if some $t \notin V$ is enabled in s then some $t' \notin V$ is in U;*

- *for every infinite execution* $s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_3 \ldots$ *and each* $a \in V$ *there is an* $i \geq 0$ *such that* a *is in the stubborn set used at* s_i*;*

Then U *is a* LTL preserving stubborn set.

Theorem 7 *If* ϕ *is a formula in LTL that does not contain the* **X** *operator, and the set* V *of visible actions contains all actions that can possibly alter the value of elementary propositions in* ϕ*, then the use of LTL preserving stubborn sets assures that* ϕ *holds of the reduced system if and only if* ϕ *holds of the original system.*

CTL* preserving stubborn sets are yet another extension in the line of path oriented stubborn sets. In addition to the preservation of paths, CTL* preserving stubborn sets need to preserve possible choices (branches) in the transition system (CTL* is a branching time temporal logic). This problem has been solved by the strong requirement that a stubborn set U consists either of a single enabled action which is invisible, or contains all enabled actions.

Definition 17 *Let* V *be a set of actions (the set of visible actions). Let* U *be a strong stubborn set in* s *where no transition is ignored, and* U *contains either a single enabled action which is not in* V*, or* U *contains all enabled actions in* s*. Then* U *is called a* CTL* preserving stubborn set.

Theorem 8 *Let* ϕ *be a CTL* formula. If a reduced state space is constructed using CTL* preserving stubborn sets in each state where* V *includes the set of transitions that can possibly alter elementary propositions occurring in* ϕ *then* ϕ *holds of the reduced transition system if and only if* ϕ *holds of the original transition system.*

At this point, we close the list of selected path oriented stubborn sets and turn to state oriented stubborn sets.

Consider some set M of states (i.e. a state property). We would like to preserve M in the sense that the reduced transition system contains elements of M if and only if the original transition system does. For a particular state s this means that if a state s' is reachable from s then at least one sequence of actions leading from s to $s'' \in M$ should be part of the reduced state space.

Return to the principle of weak stubborn sets. It assures that if there is a path from s to s' in the original state space, and that path contains actions

from the weak stubborn set in U, then s has at least one successor state in the reduced transition system that is closer to s'. All we need to assure in order to preserve M is that every path from a current state s to an element in M contains at least one action that is in the stubborn set used at s. Then, for each state in the reduced state space, another state that is closer to M, is contained in the reduced state space as well. This means finally that the closest elements of M are members of the reduced state space. As a suitable set of actions to be part of stubborn sets, we consider UP sets and their counterparts, DOWN sets.

Definition 18 (UP set, DOWN set) *Let M be a set of states, and $s \notin M$, $s' \in M$ states. Then a set U of actions is an UP set from s to M iff every execution leading from s to M contains an element of U. A set U' of actions is a DOWN set from s' out of M iff every path from s' to a state not in M contains an element of U'.*

For states in M, UP sets are not defined, while for states not in M, DOWN sets are not defined. With respect to union, intersection, and complement, UP and DOWN sets behave as follows.

Theorem 9 (Set operations versus UP and DOWN sets) *Let s be a state, M_1 and M_2 sets of states. If a set U of actions is an UP set from s to M then U is a DOWN set from s out of $\overline{M}$. Reversely, if U is a DOWN set from s out of M, then U is an UP set from s to $\overline{M}$.*

If U_1 is an UP set from s to M_1, and U_2 is an UP set from s to M_2, then $U_1 \cup U_2$ is an UP set from s to $M_1 \cup M_2$. U_1 is an UP set from s to $M_1 \cap M_2$ for $s \notin M_1$, and U_2 is an UP set from s to $M_1 \cap M_2$ if $s \notin M_2$. Analogously, if U_1 is a DOWN set from s out of M_1, and U_2 is a DOWN set from s out of M_2, then $U_1 \cup U_2$ is a DOWN set from s out of $M_1 \cap M_2$, U_1 is a DOWN set from s out of $M_1 \cup M_2$ for $s \in M_1$, and U_2 is a DOWN set from s out of $M_1 \cup M_2$ if $s \in M_2$.

The set A of all actions is always an UP or DOWN set, but not useful for reduction since we want to use UP sets as follows:

Theorem 10 (UP sets preserve M) *Let M be a set of states. Let a reduced transition system be constructed as follows. In a state $s \notin M$, use a weak stubborn set that contains an UP set from s to M. In a state $s' \in M$,*

use the empty set of actions as stubborn set. Then the reduced transition system contains states in M if and only if the original transition system contains states in M.

Proof. If the original transition system does not contain states in M then the reduced transition system cannot contain states in M since it is an underapproximation. Assume the original transition system contains a state in M but the reduced transition system does not. Then there is a path from an initial state to a state $s^* \in M$. Since this initial state is part of the reduced transition system, there are elements of the reduced state space from which s^* is reachable. Choose s^*, a state s in the reduced transition system, and a sequence of actions w such that $s \xrightarrow{w} s^*$ and the length of w is minimal. Since s^* itself is not in the reduced transition system, the length of w is greater than one. By definition of UP sets, w contains an action in U for every UP set from s to M. Since some UP set from s to M is part of the stubborn set used in s, w contains actions in the stubborn set used in s. Thus, we can rewrite w as $w_1 a w_2$ where w_1 is a sequence of actions not contained in the stubborn set used in s, and a is in the stubborn set used in s. By the principle of weak stubborn sets, $s \xrightarrow{w=w_1 a w_2} s^*$ implies that there is a state s_1 such that $s \xrightarrow{a} s_1 \xrightarrow{w_1 w_2} s^*$. Since a is in the stubborn set used in s, s_1 is part of the reduced transition system. The sequence $w_1 w_2$ leads from s_1 to s^* and is shorter then the sequence $w = w_1 a w_2$ leading from s to s^*, in contradiction to the assumed minimality of w. Consequently, the assumption that the reduced state space does not contain elements of M must be false. ♢

Thm. 10 can be immediately instantiated to a stubborn set approach for various reachability properties.

Corollary 1 (Stubborn sets for reachability properties) *Let ϕ be a state property. If a reduced transition system is constructed using in every state s a weak stubborn superset of an UP set from s to $\{s \mid s \models \phi\}$, then the reduced transition system satisfies $\mathbf{EF}\phi$ if and only if the original transition system satisfies $\mathbf{EF}\phi$. If a reduced transition system is constructed using in every state s a weak stubborn superset of a DOWN set from s out of $\{s \mid s \models \phi\}$, then the reduced transition system satisfies $\mathbf{AG}\phi$ if and only if the original transition system satisfies $\mathbf{AG}\phi$.*

As an example, Tab. 5.1 lists possible UP and DOWN sets for the elementary Petri net atomic propositions offered in our tool LoLA.

Table 5.1: UP and DOWN sets from a Petri net marking m to $\{m' \mid m' \models \phi\}$, for elementary state properties ϕ; k is a natural number

ϕ	UP set from m to $\{m' \mid m' \models \phi\}$	DOWN set from m out of $\{m' \mid m' \models \phi\}$
$p < k$	$p^\bullet$	${}^\bullet p$
$p \leq k$	$p^\bullet$	${}^\bullet p$
$p > k$	${}^\bullet p$	$p^\bullet$
$p \geq k$	${}^\bullet p$	$p^\bullet$
$p = k$	${}^\bullet p$, if $m(p) < k$; $p^\bullet$, if $m(p) > k$	${}^\bullet p \cup p^\bullet$
$p \neq k$	${}^\bullet p \cup p^\bullet$	${}^\bullet p$, if $m(p) < k$; $p^\bullet$, if $m(p) > k$

Further elementary properties could be added without major difficulties. UP sets for boolean combinations of properties can be derived from Thm. 9.

The preservation of deadlock states in the basic stubborn set approach can be verified by viewing basic stubborn sets as an instance of the UP set approach. Let M be the set of deadlock states and s be a state that is not a deadlock state. This means that there is at least one enabled action a in s, and on every execution from s to a state in M, a becomes disabled, whether by its own occurrence, or by occurrence of another transition that alters the enabling condition for a. Thus, for some enabled action a, the set of all actions that can potentially disable a, is an UP set from s to M.

The basic stubborn set approach requires only the existence of a key action. The main property of a key action is that no sequence of actions outside the stubborn set can disable it. In other words, the basic stubborn set does contain an UP set from s to the set of all deadlock states, since at least one element of the basic stubborn set must be executed before an enabled key action in the stubborn set used in s can be disabled.

For finite state systems, the UP set approach can be extended to home properties **AGEF**ϕ (for a state predicate ϕ) and the related properties **EFAG**ϕ, **AGEFAG**ϕ, **EFAGEF**ϕ.

As mentioned in Sec. 4.4.2, properties where longer alternating sequences of **AG** and **EF** precede a state predicate ϕ are equivalent to one of these four (classes of) formulas, and all these properties relate to terminal strongly connected components of the transition system:

- **AGEF**ϕ holds iff every terminal strongly connected component of the transition system contains a state that satisfies ϕ;
- **EFAG**ϕ holds iff there is a terminal strongly connected component of the transition system where every state satisfies ϕ;

- **AGEFAG**ϕ holds iff all states of all terminal strongly connected components of the transition system satisfy ϕ;
- **EFAGEF**ϕ holds iff there is a terminal strongly connected component of the transition system that contains a state satisfying ϕ.

Home properties can be verified by a two-step approach. In the first step, we compute a reduced transition system using strong stubborn sets such that the reduced transition system does not contain ignored transitions. By Thm. 5, this reduced system contains at least one state of every terminal strongly connected component of the original transition system. We pick one element from every terminal strongly connected component of the reduced transition system. It is easy to see that the set of chosen states contains at least one member of each terminal strongly connected component of the original transition system. The next lemma shows that, among the picked states, there are no elements of non-terminal strongly connected components of the original system.

Lemma 1 (Terminal scc are contained in terminal scc) *If a reduced transition system is constructed using strong stubborn sets, and no action is ignored, then every terminal strongly connected component of the reduced transition system is a subset of a terminal strongly connected component of the original transition system.*

Proof. Assume a terminal strongly connected component C_R of the reduced system is a subset of a non-terminal strongly connected component C_O of the original system (as a strongly connected set, C_R cannot be spread over more than one component of the original system). From C_O, at least one terminal strongly component C_T of the original system must be reachable. Hence, for every state in C_R, states in C_T are reachable. Choose a state $s \in C_R$, a sequence of actions w, and a state $s' \in C_T$ such that $s \xrightarrow{w} s'$, and the length of w is minimal. Since C_R is contained in the non-terminal C_O, the length of w is greater than zero. Assume, w contains elements of the strong stubborn set U used in s. Then w can be represented as $w_1 a w_2$ where $a \in U$, and w_1 does not contain elements of U. By the principle of weak stubborn sets, we have $s \xrightarrow{a w_1 w_2} s'$ which means that the a-successor of s has a shorter execution sequence to s'. Since C_R is terminal, the a-successor of s is in C_R. Consequently, we have a contradiction to the assumed minimality of w, and the assumption that w contains elements of U must be false.

Since U is a strong stubborn set, every enabled action in U is a key action. Hence, for every action $a \in U$ that is enabled in s, there are states $s_1 \in C_R$ and $s'_1 \in C_T$ such that $s \xrightarrow{wa} s'_1$, and, by the principle of weak stubborn sets, $s \xrightarrow{a} s_1 \xrightarrow{w} s'_1$. Thus, w is executable in every successor state of s in the reduced system. This argument can be applied to the successors of s, and, iteratively, to all states of C_R. w can therefore be executed in every state of C_R. This means that the first action of w is enabled in every state of C_R, but not contained in the stubborn set of any state in C_R. Consequently, the first action of w is an ignored action, in contradiction to the assumption on the reduced transition system. $\diamondsuit$

In the second step, for each picked state s separately, we check whether states satisfying ϕ (for $\mathbf{AFEF}\phi$ or $\mathbf{EFAGF}\phi$) or $\neg\phi$ (for $\mathbf{EFAG}\phi$ or $\mathbf{AGEFAG}\phi$) is reachable from s. For this purpose, we construct a reduced transition system with s being the initial state, and use the UP set approach for reachability properties. The original system satisfies $\mathbf{AGEF}\phi$ if and only if a state satisfying ϕ is reachable from every picked state. The original system satisfies $\mathbf{EFAG}\phi$ if and only if there is a picked state s from which no state satisfying $\neg\phi$ is reachable. The original system satisfies $\mathbf{AGEFAG}\phi$ if and only if for none of the picked states, a state not satisfying ϕ is reachable. The original system satisfies $\mathbf{EFAGEF}\phi$ if and only if there is a picked state s from which a state satisfying ϕ is reachable.

All four claims are immediate consequences of the mentioned relation between the two properties and terminal strongly connected components, and the fact that the set of picked state covers all terminal strongly components of the original transition system.

The UP set approach can be used for the verification of $\mathbf{EF}\phi$ and $\mathbf{AG}\phi$ ($\equiv \neg\mathbf{EF}\neg\phi$) where ϕ is an arbitrary CTL property. Since in CTL every temporal operator is immediately preceded by a path quantifier, every CTL property can be associated with a set of states satisfying it. Thus, our approach can be used as soon as we can find UP and DOWN sets for state sets associated to arbitrary CTL formulas. Some of our proposals for UP and DOWN sets depend on witness or counterexample paths for ϕ. These paths are indeed available, at least if we apply a recursive algorithm as sketched in Sec. 4.4.3. There, ϕ is explored in a state s before the successors of s in the search for $\mathbf{EF}\phi$ are computed.

As another tool for computing UP and DOWN sets, we need the concept of *dependency* and *independency*.

Definition 19 (dependent/independent action) *Two actions a and b are independent in a transition system iff for all states s, s_1, s_2 in the transition system*

1. *in the transition system, $s \xrightarrow{a} s_1$ implies that b is enabled in s if and only if b is enabled in s_1;*

2. *$s \xrightarrow{ab} s_1$ and $s \xrightarrow{ba} s_2$ implies $s_1 = s_2$.*

Two actions that are not independent, are dependent.

Informally, independent actions cannot enable or disable each other, and their order of occurrence does not alter the resulting state.

Denote with $D(a)$ the set of all actions that are dependent on a. For a sequence w, denote with $D(w)$ the set of all actions that are dependent on an action occurring in w.

We start the discussion of UP and DOWN sets with $\phi \equiv \mathbf{EF}\phi'$. If a state s does not satisfy $\mathbf{EF}\phi'$ then no state s' reachable from s can satisfy $\mathbf{EF}\phi$ since otherwise the sequence from s to s' could be linked to the witness path for $\mathbf{EF}\phi$ in s' and would thus build a witness path for $\mathbf{EF}\phi$ in s. Thus, we can use $\emptyset$ as an UP set from s to $\mathbf{EF}\phi$. As a DOWN set, we propose a more involved construction.

Theorem 11 (DOWN set of $\phi \equiv \mathbf{EF}\phi'$) *Let s be a state that satisfies $\mathbf{EF}\phi'$. Let w be a witness path for $\mathbf{EF}\phi'$ in s, i.e. $s \xrightarrow{w} s'$ for some state s' that satisfies ϕ'. Let A^- be a DOWN set from s' out of $\{s^* \mid s^* \models \phi'\}$. Then $D(w) \cup A^-$ is a DOWN set from s out of $\mathbf{EF}\phi'$.*

Proof. We need to show that without occurrence of actions in the DOWN set, $\mathbf{EF}\phi$ remains true, i.e. for every state s_1 and every sequence w' such that $s \xrightarrow{w'} s_1$ and w' does not contain elements of $D(w) \cup A^-$, $s_1 \models \mathbf{EF}\phi'$.

First, since all actions in w are pairwise independent of actions in w' (w' does not contain members of $D(w)$), we have that w is executable in s_1, w' is executable in s', and there is a state s'_1 such that $s_1 \xrightarrow{w} s'_1$ and $s' \xrightarrow{w'} s'_1$. Since w' does not contain elements of A^-, which is a DOWN set from s' out of ϕ', execution of w' in s' cannot invalidate ϕ'. Thus, s'_1 satisfies ϕ' and w, executed in s_1, is a witness path for $\mathbf{EF}\phi$ in s_1. Consequently, $D(w) \cup A^-$ is a DOWN set from s out of $\mathbf{EF}\phi'$. $\diamondsuit$

Next, we discuss UP and DOWN sets for $\mathbf{EG}\phi$. Assume that a state s does not satisfy $\mathbf{EG}\phi'$ and consider two cases $s \models \phi'$ and $s \not\models \phi'$. In case $s \models \phi'$, every path leading to a state s' where $s' \models \mathbf{EG}\phi'$ must lead through a state s_1 where $s_1 \not\models \phi'$ since otherwise the sequence from s to s' can be linked to a witness path for $\mathbf{EG}\phi'$ in s' and builds a witness path for $\mathbf{EG}\phi$ in s. Consequently, a DOWN set from s out of ϕ' is as well an UP set from s to $\mathbf{EG}\phi'$ if $s \models \phi'$ In the second case, $s \not\models \phi'$, every path leading to a state satisfying ϕ' must, in the first place, lead to a state that satisfies ϕ'. Thus, an UP set from s to ϕ' is, in this case, as well an UP set from s to $\mathbf{EG}\phi$.

For DOWN sets out of $\mathbf{EG}\phi'$, we use a similar technique as for DOWN sets out of $\mathbf{EF}\phi'$.

Theorem 12 *Let s be a state that satisfies $\mathbf{EG}\phi'$. Let $a_0 a_1 a_2 \dots a_n$ be a witness path for $\mathbf{EG}\phi'$ in s, i.e. $s = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \dots s_{n-1} \xrightarrow{a_{n-1}} s_n \xrightarrow{a_n} s_j$ for some state $j \in \{0, \dots, n\}$ where all s_i $(0 \leq i \leq n)$ satisfy ϕ'. Let A_i^- be a DOWN set from s_i out of $\{s^* \mid s^* \models \phi'\}$. Then $D(w) \cup \bigcup_{i=0}^{n} A_i^-$ is a DOWN set from s out of $\mathbf{EG}\phi'$.*

Proof. Let w' be a sequence that does not contain elements of the proposed DOWN set. Since $D(w)$ is part of the DOWN set, all elements of w' are pairwise independent of the elements of w. Thus, there are states $s'_0, \dots, s'_n$ such that $s_i \xrightarrow{w'} s'_i$ for $0 \leq i \leq n$, $s'_i \xrightarrow{a_i} s'_{i+1}$, for $0 \leq i \leq n-1$, and $s'_n \xrightarrow{a'_n} a'_j$. Since w' does not contain elements of any DOWN set from s_i out of ϕ', every s'_i $(0 \leq i \leq n)$ satisfies ϕ'. Thus, $a_0 \dots a_n$ is a witness path for $\mathbf{EG}\phi'$ in s'_0 which is the result of executing w' in $s = s_0$. $\diamondsuit$

For $\mathbf{E}(\phi_1 \mathbf{U} \phi_2)$, we can combine the consideration for $\mathbf{EG}\phi_1$ (ϕ_1 holds ...) with those for $\mathbf{EF}\phi_2$ (... until ϕ_2).

Thus, we can leave it to the reader to verify that, given an UP set A_1^+ from s to ϕ_1, an UP set A_s^+ from s to ϕ_2, and a DOWN set A_1^- from s out of ϕ_2, the set $A_1^+ \cup A_2^+$ is an UP set from s to $\mathbf{E}(\phi_1 \mathbf{U} \phi_2)$, if $s \not\models \phi_1$, and A_2^- is an UP set from s to $\mathbf{E}(\phi_1 \mathbf{U} \phi_2)$, if $s \models \phi_1$. Correspondingly, assume that $a_1 \dots a_n$ is a witness path for $\mathbf{E}(\phi_1 \mathbf{U} \phi_2)$ in s, i.e. there are states $s_0, \dots, s_n$ where $s = s_0$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$ $(0 \leq i \leq n-1)$, $s_i \models \phi_1$ $(0 \leq i \leq n-1)$, and $s_n \models \phi_2$. Let A_i^- be a DOWN set from s_i out of s_i $(0 \leq i \leq n-1)$ and A_n^- a DOWN set from s_n out of ϕ_2. Then $\bigcup_{i=0}^{n} A_i^-$ is a DOWN set from s out of $\mathbf{E}(\phi_1 \mathbf{U} \phi_2)$.

We do not have a proposal for a feasible UP or DOWN set for $\mathbf{EX}\phi'$. For universally quantified formulas other than $\mathbf{AX}\phi'$, we can find UP and DOWN sets via applicable tautologies:

$$\begin{aligned} \mathbf{AF}\phi' &\Leftrightarrow \neg\mathbf{EG}\neg\phi' \\ \mathbf{AG}\phi' &\Leftrightarrow \neg\mathbf{EF}\neg\phi' \\ \mathbf{A}(\phi_1\mathbf{U}\phi_2) &\Leftrightarrow \neg(\mathbf{EG}\neg\phi_2 \vee \mathbf{E}(\neg\phi_2\mathbf{U}(\neg\phi_1 \wedge \neg\phi_2))) \end{aligned}$$

Though we were able to define UP and DOWN sets for arbitrary **X** - free CTL formulas, we do not have an alternative stubborn set approach to CTL. The UP set approach does not offer an approach to the verification of subformulas other than elementary properties or those starting with an **EF** (with the UP set approach to reachability), **AG** (with the relation between **EFAG**), or a boolean operator (with Thm. 9). Thus, our approach as such can be used only for verification in the **EF**/**AG**-fragment of CTL. This is an interesting and non-trivial fragment.

In order to extend our approach to arbitrary CTL or CTL* formulas, we need a reduction scheme that preserves **EG** and **E**(.**U**.) formulas, too. The UP set approach seems to be principally infeasible for this task. It relies essentially on a difference between a current state and a set of target state. This difference manifests itself through UP and DOWN sets. For $\mathbf{EG}\phi'$ formulas, we are looking for a path where a property ϕ' is *preserved*. So, the goal is not to enter or to leave a set of states but to stay within a set of states. UP and DOWN sets appear to be of little value for this task. However, the UP set approach can be combined with the path oriented stubborn set approach to CTL*. When verifying a CTL formula with the recursive algorithm sketched in Sec. 4.4.3, every subformula starting with a pair of temporal operator and path quantifier is verified by a separate search. For **EF** and **AG** subformulas, we can use the UP set approach, since an UP and DOWN set can be computed for all kinds of CTL subformulas, while the path oriented stubborn set approach can be used for other subformulas.

In [KV00], the UP set approach to reachability properties has been relaxed, and a new UP set related approach to home properties has been proposed, not requiring a two-step approach.

The main differences to the UP set approach presented above are:

- a global Down set is used instead of our state dependent DOWN sets. An action is in the global Down set for a set of states M, iff for every

pair of states $s \in M$ and $s' \notin M$ with $s \xrightarrow{w} s'$ (for a sequence w of actions), at least one action in w in contained in the global Down set. UP sets are used as in the approach above;

- An UP set is not required to be contained in the stubborn set of every state—it can be spread over the stubborn sets of a whole terminal strongly connected component.

Formally, they require:

Definition 20 (Relaxed state oriented stubborn sets) *(Slightly less general than [KV00]) Let s be a state in a transition system. A strong stubborn set U in s is called a* relaxed state oriented stubborn set *iff*

- *If an action in the global Down set out of M is enabled in s then an UP set from s to M is contained in U;*
- *for every terminal strongly connected component C, the root node s_C of C in depth first search, and for every member a of an UP set from s_C to M, there is at least one $s \in C$ where a is contained in stubborn set used in s.*

In [KV00], they use a slightly weaker condition than strong stubborn sets that we do not want to introduce here, for the sake of readability. Relaxed state oriented stubborn sets are generally smaller than purely UP set oriented stubborn sets, and can lead to substantially smaller state spaces. They can be used for reachability properties and, with a dedicated home property preseving condition (see [KV00] for details), for home properties. For home properties, we can alternativly use relaxed state oriented stubborn sets in connection with the two-step approach described earlier. An extension of the relaxed towards complex temporal state formulas is not in sight. A major obstacle is the use of global Down sets. Several local UP or DOWN sets we proposed earlier required knowledge of a particular counterexample or witness path. Such a path is not available for a global Down set. Thus, we lack a feasible approach to global Down sets for temporal state properties.

There is another difference between pure and relaxed stubborn sets. Using the pure UP set approach, it is guaranteed that for the shortest path leading from the initial state to a state in M, a path of same length leading from the initial state to a state in M is contained in the reduced transition system. This property is not true of relaxed state oriented stubborn sets.

5.2 Algorithms

For Petri nets, a commonly used algorithm for computing stubborn supersets of a given set of transitions is depicted in Fig. 5.3. It does not necessarily produce the smallest possible stubborn sets, but is efficient and fairly easy to implement.

Figure 5.3: Constructing a strong stubborn superset of a set of actions

```
1   procedure WeakStubSuperset(m : Marking, U: set of Transitions):
        set of Transitions
2   var U' : set of Transitions;
3   var t : Transition;
4   var p : Place;
5   begin
6     while U ≠ ∅ do
7       choose t ∈ U; U := U \ {a}; U' := U' ∪ {a};
8       if t enabled in m then
9         U := U ∪ (•t)•;
10      else
11        choose p s.t. W([p,t]) > m(p);
12        U := U ∪ •p;
13      fi
14    done
15    return U';
16  end.
```

In LoLA, an insufficiently marked place (line 11) is determined during the test for enabledness of t. We always use the first disabled place encountered during the enabling test. In [Var92], correlations between the choice of p in line 11 and the size of the resulting stubborn set is studied, unfortunately without yielding results that would strongly suggest a sophisticated procedure for choosing p. The insertion operations (lines 9 and 12) can be implemented efficiently. For every transition t, we can store explicitly the list of transitions in $(\bullet t)\bullet$, and for every place the list $\bullet p$. Then, every transition gets has a pointer that points to its own list if t is enabled and points to its disabled pre-place's list if t is disabled. This pointer is updated during

the enabling test for t. In the enabling test, an insufficiently marked place is immediately available. The actual stubborn set procedure simply builds a closure where, with every transition, the pointed list of transitions is inserted, too.

Various other conditions can be implemented similarly. In particular, requirements for visibility ("none or all visible transitions are in a stubborn set", "at least one invisible transition is in the stubborn set", "if a transition of a Down set are contained in the stubborn set, so are all transitions in an UP set", and others).

For variable based approaches, an even simpler approach is frequently used. It relies on the symmetric dependency relation introduced in the previous section. From such a relation, sets of actions meeting various requirements on stubborn sets can be computed [Pel93]. For two guarded commands, an easy sufficient criterion for independency is: two guarded commands are independent if the sets of variables occurring in the guards and assignments, are disjoint. It is obvious that such commands cannot disable or enable each other, nor alter the effect of the other command's execution. In message passing systems, local actions of components are treated as independent while send and receive actions are depend on all actions of the involved components. For Petri nets, dependency can be defined as well. Two transitions t and t' are independent iff $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$. However, for Petri nets the above stubborn set approach leads generally to better results.

One of the conditions for path oriented LTL preserving stubborn sets (for every infinite execution $s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_3 \ldots$ and each $a \in V$ there is an $i \geq 0$ such that a is in the stubborn set used at s_i) cannot be implemented in the same fashion as the conditions above. Since every infinite execution in a finite state system contains cycles, we can implement this condition by assuring that every visible transition occurs in the stubborn set of at least one state in every cycle of the reduced transition system. The easiest way of covering all cycles is to observe, as discussed in the section on depth first search, that every cycle contains at least one state that is on the depth first search stack while its predecessor state is explored. Thus, we can add all visible transitions into the stubborn set of states that have successors on the depth first search stack. Valmari [Val98] points out that there might be other possibilities to implement this condition and could lead to better reduction. However, prominent LTL verification tools like SPIN [Hol91] seem to use the

simple implementation.

UP and DOWN sets for simple properties, or boolean combinations of properties can be implemented recursively, according to the rules in Thm. 9. Some of the UP and DOWN sets for temporal properties rely on counterexample or witness paths. Given these paths, the calculation of UP and DOWN sets is straightforward, too.

Storing witness paths for every state in the transition system explicitly seems to be costly. However, all we need is one pointer per subformula in every state, linking states to witness or counterexample paths. Assignments to that pointer can be easily integrated in the model checking procedures searchEU and searchAU discussed in Sec. 4.4.3.

The core feature of those two procedures was that they determine the value of an until formula not only for the state they were started on, but also for all states that are visited during search.

If searchEU finds a witness path for the state it was started on, then all states on the search stack satisfy the formula, too, and the corresponding suffix on the stack is their witness path. Thus, for states on the stack, we can let the witness path pointer point to their successor on the stack. States not on the stack (in the moment where searchEU finds a witness) that are members of a strongly connected component not yet closed, satisfy an existential until formula, too. For those, a witness path would consist of a path to any state of the search stack, concatenated with that state's witness path (remember that all states in the search space satisfy the left subformula of the checked until formula). For states that left the search stack, a path to the stack can be built by letting every state point to the one successor state that realizes the current state's lowlink value in Tarjan's algorithm on page 36. This can be the successor from which a state inherits the lowlink value in line 18, or the successor with small dfs number in line 14. All those successors are in the same strongly connected component as the considered state. Furthermore, the lowlink value promises reachability of a state n' with that value as dfs number. Inheriting a lowlink value in Tarjan's algorithm means that n' can be reached via the successor state from which the value is inherited. Let n be a state that is in a component not yet closed, but not on the search stack. n's lowlink value is smaller than its dfs number since otherwise it would be the root state of its component which contradicts the assumption that the component is not yet closed. Thus, following the path described above leads eventually to a state with smaller dfs number than n.

This process cannot be continued indefinitely since dfs numbers are natural numbers. Thus, the described path must eventually reach a state on the search stack. This is what we wanted to achieve.

The situation for searchAU is easier. If searchAU finds a counterexample path, then all states on that path violate the universal until formula while all states the have been visited but already closed do not satisfy the until formula. Thus, all we need to do is to link the states on the counterexample path to their successor. Then, the path starting at any member of the counterexample path is a counterexample for that state.

Since witness and counterexample paths for nextstep formulas are trivial, and the remaining CTL operators can be traced back to either **E** (. **U** .) or **A**(.**U**.), we have a data structure that can be updated on-the-fly during explicit CTL model checking, and from which we can retrieve witness and counterexample paths for arbitrary CTL formulas. This technique is implemented in LoLA's CTL model checking procedure.

Finally, we study an efficient implementation of a strategy to avoid ignored actions. According to [Val91b], ignorance can be avoided as follows:

1. Combine depth first search with detection of terminal strongly connected components;

2. Upon detection of a terminal strongly connected component, check if there are ignored actions in this component;

3. If so, add a stubborn superset of one ignored action to the stubborn set used in the the root state of the detected component and explore new enabled actions.

The new actions do not alter the depth first nature of exploration since the search algorithm behaves exactly as it would have behaved in the case where the new actions were present from the beginning. For detecting terminal strongly connected components, we proposed an efficient solution in Sec. 4.4.2. So, the remaining implementation issue here is to find out whether a terminal component contains ignored actions. The definition suggests that a second run through all states of the component be necessary which is fortunately not the case. Thus, implementations of the state space that do not store states and lists of fired/enabled transitions explicitly, can be used in connection with the construction of ignorance free transition systems.

In fact, all we need is to store two numerical values for each action. The first value, $lastfired(t)$ is the largest dfs number of a state s where t was an enabled action in the stubborn set used in s. $lastdisabled(t)$ is the largest dfs number of a state where t was disabled. Upon detection of a terminal strongly connected component, we know that the members of that component are exactly those states whose dfs number is greater or equal than the root state of that component (the state that is current state at the moment of detecting the component). Thus, t is enabled in all members of the component if and only if $lastdisabled(t)$ is smaller than the dfs number of the current state, and t has never been fired in the component iff $lastfired(t)$ is smaller than the root state's dfs number. Consequently, t is ignored in the detected component iff both $lastdisabled(t)$ and $lastfired(t)$ are smaller than the root state's dfs number.

Conditions such as the requirement that elements of an UP set occur at least once in a terminal strongly connected component (for relaxed state oriented stubborn sets) can be implemented similarly.

5.3 Performance

The first tables compare the UP set approach with the relaxed UP set approach for reachability properties and full state space generation. Our results show that the behavior of the two methods depends largely on whether or not the tested predicate is satisfiable. Table 5.2 shows results for unsatisfiable predicates while Table 5.3 refers to satisfiable predicates. In the second table, state spaces refer to the portion of the state space that has been constructed before the first satisfying state has been found (on-the-fly). There, we report additionally the length of the witness path (P) leading to the found state. As unsatisfiable properties, we took mutual exclusion between two neighbors in the philosophers examples, and mutual exclusion between two writing processes in the DA examples. As a satisfiable property for the PH examples, we checked the possibility for all but the first philosopher to hold their right forks just before finishing their cycles. There is only one state satisfying this property, and that state is the most distant one from the initial state in the whole system. In the DA examples, we checked for reachability of states where all readers are reading at the same time.

For the PHn systems, the relaxed stubborn set approach shows its full power. Systems as DAn where both methods lead to the same reduction,

Table 5.2: Reduction for reachability of unsatisfiable predicates

	PH12	PH13	PH400	DA19	DA100	DA500
states (full):	531440			524307		
edges (full)	4251516			9961510		
time (full)	25.3			69.7		
states (UP)	254911	732277		39	201	1001
edges (UP)	433258	1248253		76	400	2000
time (UP)	2.9	9.4		0.032	0.084	106.81
states (relaxed)	398	470	478802	39	201	1001
edges (relaxed)	540	637	638800	76	400	2000
time (relaxed)	0.025	0.028	48.312	0.04	0.78	101.6

seem to be exceptions. At least, several other experiments lead to similar results as for the PH systems. The extraordinarily large run time for the DA500 system is not caused by state space generation. Most of the time is spent for generating the internal representation of the system with its more than half a million arcs.

Table 5.3: Reduction for reachability of satisfiable predicates

	PH20	PH500	PH2000	DA19	DA100	DA500
states (full)	118074			266892		
edges (full)	161765			1709888		
time (full)	1.35			11.6		
path (full)	118073			266891		
states (UP)	126	3486	13986	20	101	501
edges (UP)	125	3485	13985	19	100	500
time (UP)	0.026	0.61	19.1	0.031	0.7	82.2
path (UP)	125	3485	13985	19	100	500
states (relaxed)	145	3505	14005	21	102	502
edges (relaxed)	145	3505	14005	21	102	502
time (relaxed)	0.026	0.64	20.0	0.035	0.72	85.4
path (relaxed)	141	3501	14001	19	100	500

For satisfiable predicates, the reduction techniques rank in reverse order. This can be explained by a greedy heuristics that is inherent in UP sets. These sets contain actions that *must* occur at some point before a target state can be reached. In LoLA, UP set transitions are the first ones to be explored. Thus, those transitions that must occur anyway at some point, are explored early. This way, there is an increased probability that target states are approached early. In the UP set approach, every state contains

UP set actions while in the relaxed technique, only states enabling Down set actions contain the full UP set. Otherwise, UP sets can be spread over a whole terminal strongly connected component. Thus, the mentioned greedy heuristics is more influential in the pure UP set approach. This explains the differences in the witness path lengths, and the state space sizes. Without immediate termination at a satisfiable state, state space sizes should be comparable to those of unsatisfiable properties where the on-the-fly character of our implementation has no effect.

All methods behave significantly better on satisfiable properties. Given further that the relaxed method performs significantly better on unsatisfiable properties than pure UP sets, and only little worse on satisfiable properties, this is the one we recommend for implementation. The pure UP set method may produce smaller witness paths. When witness paths are used for diagnostic purposes, we usually want to have paths as small as possible. In such situations, one could think of a scenario where the relaxed technique is used for deciding satisfiability, and, in the satisfiable case, the pure UP set technique is run subsequently for computing a possibly smaller witness path. As another advantage, the pure UP set method does not require detection of terminal strongly connected components which gives more flexibility for implementation.

For home properties, we proposed two techniques. Using the first technique (in the tables below referred to as RS/I), we compute first a set of representatives of all terminal strongly connected components (phase 1), and check then if, from all of these representatives, states satisfying the property exist (phase 2). For phase two, we can use either UP sets or relaxed stubborn sets. We report the number of states and edges in both phases separately (St I, St II, Ed I, Ed II) since the state space of phase one can be removed before starting phase 2. Time corresponds to overall run time.

The second technique (in the tables referred to as HS) is the dedicated home property preserving version of relaxed stubborn sets in [KV00].

In Table 5.4, we report results for several properties on the PHi systems. *All have left* is true of the state where all processes have taken their left forks (this is the deadlock state). This property is global (refers to a large number of places) and is true (it is always possible to reach the deadlock state). *All but one have right* is true of the state where all but the first process have exited their critical sections, returned their left forks, but not yet returned their right forks. This is another global property, but it is not a

home property since it is violated by the deadlock state (a singleton terminal strongly connected component).*First has left* is true of every state where the first process has its left fork, but not the right one. This is a local property (refers to only one place) and is a home property.

Table 5.4: Home properties for PHi system

	all but one have right			all have left			first has left		
	PH100	PH400	PH15000	PH12	PH18	PH400	PH12	PH100	PH400
HS St:	101	401	15001	9983	497812		398	29702	478802
HS Ed:	100	400	15000	15989	786212		540	39700	638800
HS t:	0.06	0.22	104	0.12	6.35		0.03	0.8	43.3
RS/I St I:	29702	478802		398	930	478802	398	29702	478802
RS/I Ed I:	39700	638800		540	1242	638800	540	39700	638800
RS/I St II:	1	1		1	1	1	1	1	1
RS/I Ed II:	0	0		0	0	0	0	0	0
RS/I t:	0.88	44.3		0.03	0.04	44.3	0.03	0.9	44.3

Table 5.5 refers to DAi systems. In contrast to the PHi systems, the whole state space is strongly connected, so the only terminal strongly connected component is the whole state space itself. We verified the properties *two are writing* (true of every state where the first two processes are writing, i.e. never satisfiable), *all readers reading* (true of the states where all processes with read access are reading), and *first reading* (true of every state where process 1 is reading). The second property is global, the other two properties are local properties.

Table 5.5: Home properties for DAi system

	two are writing			all readers reading			first reading		
	DA20	DA100	DA500	DA20	DA300	DA500	DA50	DA100	DA500
HS St:	41	201	1001	837	100517		101	201	1001
HS Ed:	80	400	2000	1691	201331		200	400	2000
HS t:	0.04	0.81	105	0.06	302		0.13	0.82	105.5
RS/I St I:	41	201	1001	41	601	1001	101	201	1001
RS/I Ed I:	80	400	2000	80	1200	2000	200	400	2000
RS/I St II:	41	201	1001	22	302	502	4	4	4
RS/I Ed II:	80	400	2000	41	601	1001	5	5	5
RS/I t:	0.05	0.93	129	0.05	24	108	0.14	0.83	105

These results lead us to the following conclusions. First, if the property is local, the dedicated home state preserving method produces as many states

as the first phase of the two phased method. That is, for local properties the dedicated method is the clear winner. For global properties, the dedicated method may outperform the two phased method if it runs early into a terminal strongly connected component not containing a satisfying state (see Table 5.4 with the "all but one have right"property). In that case, the dedicated method can terminate immediately while the two phased method has to complete phase one before checking any component. Thus, the home state preserving method has a better on-the-fly behavior when the verified property is not a home property and the system has a nontrivial component structure (i.e. has more than one component). If, however, the whole state space is strongly connected (which is often the case for reactive systems), or the property to be verified is indeed a valid home property, the two phased method outperforms the HSPP method on global properties. These performance patterns repeated on several other systems we experimented with.

Our experimental evidence suggests that both methods be offered in a verification tool. The on-the-fly behavior of the two phased method could be improved by running phase two in parallel to phase one (on a different computer). This way, phase two could already start as soon as the first terminal strongly connected component is detected by phase one.

Our data show substantial state space reduction even on global properties where competing techniques fail. For local reachability properties, many existing techniques are applicable and it would require a large amount of experimental data to find out all the advantages and disadvantages of each of them. For local home properties there is only one competing stubborn set approach—the CTL*-preserving stubborn sets of [GKPP95]. ϕ is a home state property if and only if the CTL formula **AG EF** ϕ holds of the system. On all the local properties reported above, the relaxed UP set method outperforms LoLA's CTL model checking algorithm with CTL*-preserving stubborn set reduction [GKPP95]. Among the experiments not reported here, there was only one case where the model checking algorithm produced less states than the home state preserving technique (203036 vs. 234585). That is, even on local home properties, UP set based methods are at least competitive with path oriented stubborn set methods.

5.4 Compatibility

At this stage, stubborn sets are the only reduction technique we introduced. Thus, we can study compatibility only with respect to different search strategies.

We can divide the different stubborn set approaches into two classes. The first class (simple stubborn set approaches) contains stubborn set approaches where the system description, the property to be verified, and the current state are sufficient to determine a stubborn set. The second class (advanced stubborn sets) contains approaches where, in addition, knowledge about cycles or strongly connected components is necessary. Basic stubborn sets and the pure UP set approach are simple stubborn set approaches while all other path oriented methods as well as relaxed state oriented stubborn sets are advanced methods.

Simple stubborn set approaches can be used in connection with arbitrary search strategies, including breadth first search and distributed search. Simple stubborn sets can be determined without knowledge about mutual relations between states. The only change necessary in the original search algorithm is to replace the set of all enabled transitions by the set of all enabled transitions in a computed stubborn set.

Today, advanced stubborn set methods cannot be used in connection with breadth first search or distributed search (at least, not with the distribution scheme proposed in this thesis). To the author's best knowledge, only depth first search is related to strongly connected components closely enough to detect components while constructing the state space. Methods to detect strongly connected components by other methods than depth first search appear to be prohibitively inefficient.

Since, among the classes of properties listed in Sec. 1.1.2, only reachability properties can be verified using simple stubborn set techniques, incompatibility with distributed search is a severe restriction.

5.5 Discussion

Stubborn sets and similar methods promise remarkable state space reduction for concurrent systems. Concurrency is the most important source of commutativity between actions as required for stubborn set methods. Concurrency contributes, at the same time, substantially to the state space explosion.

Thus, stubborn sets directly combat a major source of state explosion.

We described many existing stubborn set methods as a consistent line of approaches that are all based on preserving paths with desired properties. All these approaches share some features, such as the distinction between visible and invisible actions. With the UP set approach, we have proposed a new family of stubborn set methods, focused on states rather than on paths. It complements the path oriented family of stubborn set methods through its capability of achieving substantial reduction for global properties. Our UP set approach is not the first state based approach.

In [Val93], a stubborn set method is proposed where the property to be verified is coded as a tester process with accepting states which is synchronized with the actual system. In this setting, state oriented stubborn sets (focused on reaching accepting states in the tester process) have been used first (not counting the state oriented view on basic stubborn sets). The new feature of the UP set approach presented here is that state oriented stubborn sets can be used on the original system, without constructing an extension of the system for coding the property.

Open problems connected to stubborn set methods include a way to apply advanced stubborn set approaches in connection with breadth first or distributed search algorithms. For this purpose, it seems to be necessary to find a completely different way for avoiding ignored actions, not relying on knowledge about terminal strongly connected components.

Other future contributions could include more dedicated stubborn set approaches to small, but frequently occurring classes of properties. The larger the class of preserved properties, the larger are obviously the restrictions for state space reduction, and the larger the reduced transition system will be.

Interesting classes of properties that still deserve dedicated stubborn set approaches, include immortality properties ($\mathbf{GF}\phi$), stabilization properties ($\mathbf{FG}\phi$), goal properties ($\mathbf{F}\phi$), and lead-to properties ($\mathbf{G}(\phi_1 \longrightarrow \mathbf{F}\phi_2)$). All these properties are currently covered only by general LTL preserving stubborn sets.

The stubborn set approach makes only few assumptions on the system description formalism. The only required feature is that it must be possible to isolate actions and their dependency on other actions. In variable based approaches, the easiest way is to define that two actions are independent iff the sets of variables involved in both actions is disjoint.

In systems that do not exhibit significant concurrency, stubborn set ap-

proaches usually lead to only insignificant reduction. This applies, for instance, to real-time systems, where a straightforward application of stubborn set method leads to the following situation: Since enabledness of action depends on real-time constraints, a "clock tick"needs to be considered as a separate action. It turns out, that in all reasonable examples, the tick event is in virtually all states member of the stubborn set and causes all remaining actions to be inserted. Thus, no reduction can be achieved.

Recently, progress has been made in applying stubborn sets in the real-time domain [Min99, BJLY98]. The key idea is to assign local clocks to system components that run asynchronously unless components interact. Then, a clock synchronization is performed prior to the interaction. This way, the strong conditions imposed by a global clock on stubborn sets can be relaxed. We believe that similar approaches could work for other domains where large systems are made of component but stubborn sets do not work well due to the used synchronization mechanism.

Chapter 6

Symmetries

There are at least two major sources of symmetries in systems. The first one is related to data. Executions starting from a state where some variable has value a, often correspond closely to executions where the same variable has another value b. An extreme case of such situation occurs in many data transfer protocols where the actual data are pure "passengers" with no influence on the control flow. The second source of symmetries is the replication of components in the design of larger systems. Hardware systems, as well as distributed systems, often include a number of identical components, wired by a more or less regular communication network. For every state that can be reached in the system, another state can be reached as well where the components just exchange roles. Symmetry reduction is about exploring only one of principally equivalent states yet being able to infer from the explored subspace the behavior of the complete state space.

Three issues are related to symmetry reduction. First, we need a technology to gather information about symmetries existing in the given system. Second, we need techniques to apply this information in the construction of a reduced system. Third, we need to establish relations between properties in the reduced system and properties in the complete system.

For gathering information about symmetries, there exist two major approaches, somehow related to the two major sources of symmetry. The first approach is based on investigating data types present in the system. For instance, if there is a data type for which assignment to variables, and symmetric comparisons ($=$, $\neq$) are the only admitted operations, any state is basically equivalent to a state where all values of that data type that to not occur as constants in the system description, are arbitrarily permuted (all

variables get the permuted value of their original value). Clearly, for every state reachable from the original state, the corresponding permuted state is reachable from the permuted original state. Such a data type is usually referred to as *scalar set* . In presence of a scalar set, gathering information about symmetry corresponds to collecting all constants of that type in the system description, and checking, that only admitted operations are used for that type. There are tools (for instance Murϕ [DDHC92]) where symmetry reduction is solely based on scalar sets. Another data type that induces symmetry is finite intervals with successor (possibly predecessor as well), and symmetric comparisons ($=$,$\neq$) as the only admitted operations. In this context, replacing any value of that type by the nth successor leads to an equivalent state. For every state from the original state, the correspondingly "shifted"state is reachable from the "shifted"original state.

The list of data types and their induced symmetries could be extended. However, all existing tools are restricted to exploring at most these two kinds of data types (possibly with minor variations) As we shall see later, these two kinds of data types enable particularly efficient procedures to construct the reduced state space which may not be true for other data types (for instance, arrays of scalars). The overall symmetry of the system is combined from the symmetry induced by each single data type involved. This approach was first proposed in [HJJJ84] for high level Petri nets, then completely automated in [CDFH90] for a class of high level Petri nets that were syntactically constrained to the two kinds of data types mentioned above. Outside Petri nets, the scalar set approach, as used for instance in [DDHC92], is the most prominent example of an actual implementation of the symmetry method.

The second approach to gathering information about symmetries focuses on the network structure between identical components. Such a network forms a directed or undirected graph. A graph automorphism is a permutation of the nodes of the graph that preserves connectivity. That is, the permuted graph is structurally identical to the original graph. Now, permuting local states of components in accordance with a graph automorphism leads to a state that is equivalent to the original state in the same sense as mentioned in the data type approach. Gathering information about symmetry in this approach amounts to calculating a generating set of the graph automorphisms of the network. This approach was first proposed in [Sta91] and put to work in [Sch00a, Sch00b]. Both approaches work on low level Petri nets where the graph aspect is dominant and information about data types is not available. However, there is no reason why the approach would

not work for other network based formalisms. In [CEFJ96, ES96, ID96], an automorphism based approach is proposed for arbitrary transition systems in the context of model checking, but no particular implementation is reported.

Of course, reasonable systems could contain both data symmetry and network symmetry. Both approaches are capable of handling this situation by representing one aspect in terms of the other. In the data type approach, one would model the network by a data type where every value corresponds to a component, and operations are used to identify neighbors in the network. For instance, a clique network (everybody is connected to everybody else) can be expressed by a scalar set, and a ring network can be expressed by a data type that has a predecessor and/or successor operation for pointing to neighbors. Other kinds of networks are difficult to handle as long as no other data types are supported. For instance, [CFG94, Chi98] reports the application of the data type symmetry approach to a three dimensional grid network (see Fig. 6.1) where extensive modeling tricks were necessary to exhibit a tiny fraction of the symmetry (4 out of 48 possible permutations) as a ring like data type.

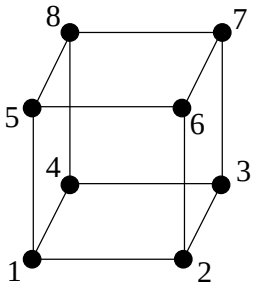

Figure 6.1: Consider this picture as a graph (assume that for all connected vertices v and v', both $[v, v']$ and $[v', v]$ are edges). This graph has 48 automorphisms.

Data dependencies can be represented as graphs. One can, for instance, replace variables by several instances each representing a possible value that variable can take. The effect of operations on variables can be represented as edges between variable instances. This is what basically happens when high level Petri nets are transformed into low level Petri nets (see an illustration in Fig. 6.2). Using graph automorphisms, symmetry of data types other than those corresponding to rings and cliques can be handled. On the other hand, a graph exhibiting the symmetry may be prohibitively large. Thus,

both symmetry approaches are important, the data type approach to handle large systems with "standard"symmetry structure, and the automorphism approach to handle systems (not quite as large) with nonstandard symmetry structure.

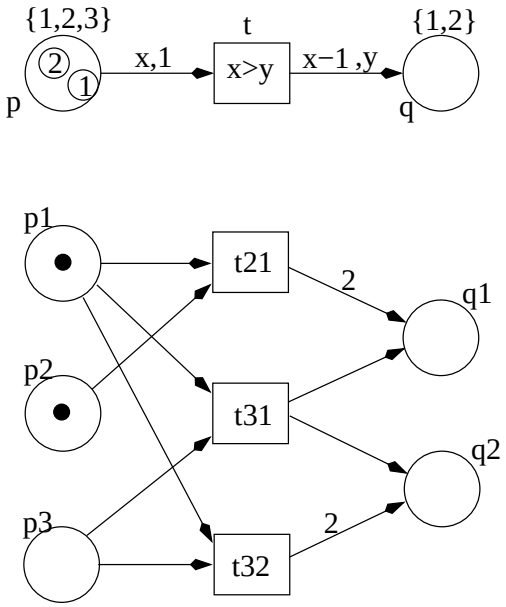

Figure 6.2: A high level net and a corresponding Petri net. In the Petri net, data related issues of the high level net have a graph representation

In the sequel, we are going to concentrate on the automorphism approach to symmetry. For studying property preservation, this is the more general approach than data type symmetries since it includes, but is not restricted to, clique and ring symmetry. For the gathering of symmetries, graph automorphism techniques are generally more involved than inspections of syntactically restricted data types. For the construction of a reduced state space, methods based on graph automorphisms include the standard method of data type symmetry, so we obtain a sufficient picture of both symmetry approaches by just looking at the graph automorphism based one.

6.1 Theory

A *group* $G = [M, \cdot, 1]$ consists of a set M, an associate binary operation $\cdot$ on M, and a neutral element 1 such that for each element $g \in M$ there is an inverse element g^{-1} holding $g^{-1} \cdot g = g \cdot g^{-1} = 1$. The inverse g^{-1} of an element g is unique. A subgroup $H = [M', \cdot|_{M'}, 1$ consists of a subset M' of M that is closed under $\cdot$ and inversion (thus, it contains the neutral element).

For a group G, and a subgroup H of G, the left and right *orbits* Hg and gH of an element $g \in G$ are defined by $gH = \{g \cdot h \mid h \in H\}$ and $Hg = \{h \cdot g \mid h \in H\}$.

For $g' \in Hg$ one can easily verify $Hg' = Hg$: Let $g^* \in Hg$. Thus, there is are $h_1, h_2 \in H$ s.t. $g^* = h_1 \cdot g$ and $g' = h_2 \cdot g$. This leads to $g^* = h_1 \cdot h_2^{-1} \cdot g'$. Since H is a subgroup, $h_1 \cdot h_2^{-1}$ is in H, so $g^* \in Hg'$. Symmetrically, $g^* \in Hg'$ implies $g^* \in Hg$. The dual property—$g' \in gH$ implies $gH = g'H$—holds as well.

From the property just proven, and the fact that $g \in Hg$ (H contains the neutral element), we can deduce first, that orbits sharing elements are equal, and second that every element of G is element of one orbit w.r.t. H. Thus, the sets of orbits w.r.t. H form a partition of G. The orbit structure of groups shall turn out to be instrumental for defining a generating set of G: every set of group elements that generates a subgroup H of G, and contains at least one element of each orbit w.r.t. H, generates G. This fact can be immediately derived from the partitioning property and the definition of orbits.

A *permutation group* operating on a finite set S consists of a set of bijective mappings on S including the identity mapping $id(x) = x$ (for all x) as neutral element, closed under the operation $\circ$ with $(\sigma_1 \circ \sigma_2)(x) = \sigma_2(\sigma_1(x))$ which is associative. A permutation group is indeed a group since for every bijection σ there is a k such that $\sigma^k = id$ which proves that every bijection in a permutation group has an inverse element σ^{k-1} which is, by closure of $\circ$, a member of the group.

Assume, for simplicity, $S = \{1, \ldots, n\}$ and consider a permutation group Π operating on S. Given some $i \in S$, define subgroups Π_i of Π by $\Pi_i = \{\sigma \mid \sigma(j) = j \text{ for } j \leq i\}$. That is, Π_i consists of all elements of Π that are the identity on the first i elements of S. It is easy to verify that Π_i is a subgroup of Π and, for $i > 1$, a subgroup of Π_{i-1}. Π_i spans orbits O_{ij} in Π_{i-1} (Π_1 in

Π, resp.) where

$$O_{ij} = \{\sigma \mid \sigma \in \Pi, \sigma(i) = j, \sigma(k) = k \text{ for } k < i\}.$$

That is, each orbit O_{ij} collects those elements of Π that are the identity up to and not including some element i, and share the same image at element i. Some of the O_{ij} are empty, at least those where $j < i$ (since images less than i are already assigned to elements less than i, and a bijection cannot assign the same image twice). So technically, only nonempty O_{ij} are actual orbits. Furthermore, we have $\Pi_{n-1} = \Pi_n = \{id\}$. That means, starting with id, and appending, for each i from $n-1$ down to 1, one element of each nonempty O_{ij}, we obtain successively generating sets for all Π_i, and finally for Π. A closer look at the definition of orbits and the generating set just defined shows that each element σ of a permutation group can be generated by a sequence of n generators

$$\sigma = \sigma_{nj_n} \circ \sigma_{n-1j_{n-1}} \circ \cdots \circ \sigma_{1j_1}$$

where σ_{ik} is the element of O_{ik} chosen to be part of the generating set. If no more than one element of each O_{ij} is chosen to be in the generating set (such a choice is possible since all O_{ij} with $j > i$ are disjoint and for O_{ii} one can always chose the identity), the representation of σ in the way described above is even unique, that is from

$$\sigma_{nj_n} \circ \sigma_{n-1j_{n-1}} \circ \cdots \circ \sigma_{1j_1} = \sigma_{nj'_n} \circ \sigma_{n-1j'_{n-1}} \circ \cdots \circ \sigma_{1j'_1}$$

it follows $j_1 = j'_1, \ldots, j_n = j'_n$. The reason is that for the smallest k where $j_k \neq j'_k$, $\sigma_{1j_1} \circ \cdots \circ \sigma_{kj_k}(k)$ is different from $\sigma_{1j'_1} \circ \cdots \circ \sigma_{kj'_k}(k)$. The other generators to be composed are the identity on at least the first k elements, so they cannot compensate this difference. In consequence, the systematic combination of n generators can be used to exhaustively and repetition free enumerating the elements of the permutation group.

Let $G = [V, E, c]$ be a labeled directed graph, i.e. a directed graph $[V, E]$ with a set V of vertices and a set $E \subseteq V \times V$ of edges, and a function $c : V \cup E \longrightarrow D$ labeling each vertex and edge with an element of some domain D.

Definition 21 (graph automorphism) *A bijection σ on V is a graph automorphism of G iff, for all $v, v_1, v_2 \in V$,*

- $c(v) = c(\sigma(v))$;

- $[v_1, v_2] \in E$ *if and only if* $[\sigma(v_1), \sigma(v_2)] \in E$*; and*
- $c([v_1, v_2]) = c([\sigma(v_1), \sigma(v_2)]$*;*

A graph automorphism permutes vertices such that the edge relation and the labeling are preserved. The set of all automorphisms of a graph G forms a permutation group. Thus, all considerations about generating sets of permutation groups apply to graph automorphism groups.

Example. Consider the automorphism group of the 3-dimensional unit cube, depicted in Fig. 6.1. Table 6.1 lists all automorphisms of this group, all relevant subgroups, orbits, and a possible generating set.

Table 6.1: Automorphisms of the 3-dimensional unit cube. Every column represents an automorphism σ. Row i contains $\sigma(i)$. Columns marked "x" are members of a generating set built as suggested in the text. Below the actual automorphisms, membership in relevant orbits and subgroups is listed.

1	1	1	1	1	1	22222233333344444455555566666677777788888							
2	2	4	4	5	5	33116644227711338866118855772288663377554							
3	6	8	3	6	8	47547518618625725727427481381354254263163							
4	5	5	2	2	4	16633127744238811318866172255763388654477							
5	4	2	5	4	2	61361372472483183181681627527536836845745							
6	3	3	8	8	6	74455781166852277572244718833145522436611							
7	7	7	7	7	7	88888555555666666333333444444111111222222							
8	8	6	6	3	3	55774466881177552244772233118822445511336							
x	x	x		x		x	x	x	x	x	x	x	
$\Pi_1 = O_{11}$						O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	
$\Pi_2 = O_{22}$			O_{24}	O_{25}									
$\Pi_3 = O_{33}$	O_{36}												

A Petri net $N = [P, T, F, W, m_0]$ can be seen as a labeled directed graph with $V = P \cup T$, $E = F$, $c(p) = m_0(p)$ for $p \in P$, $c(t) = \diamondsuit$ for $t \in T$, and $c([x, y]) = W([x, y])$ for $[x, y] \in F$. Disregarding the initial marking, we can replace the labeling on places by $c(p) = \circ$ for all $p \in P$. A graph automorphism of a Petri net is called a Petri net symmetry.

Let Σ_N be the graph automorphism group defined this way (disregarding m_0), and Σ_{Nm_0} the one regarding m_0. Obviously, Σ_{Nm_0} is a subgroup of Σ_N. In the sequel, we consider a subgroup Σ of Σ_N, that is considered arbitrary for now. Later, we shall study the impact of the choice of Σ on the preservation of certain properties. Call Σ a symmetry group.

A symmetry σ, by definition a bijection on Petri net nodes, can be extended to a mapping on markings. The idea is that tokens are moved by σ from a place p to the place $\sigma(p)$. That is, define $\sigma(m)(\sigma(p)) = m(p)$.

This notion leads to a major relation between symmetries and state spaces.

Theorem 13 (Main theorem for Petri net symmetries, [Sta91]) *Let m, m' be markings, t a transition of a Petri net N, and $\sigma \in \Sigma_N$. Then $m \xrightarrow{t} m'$ holds if and only if $\sigma(m) \xrightarrow{\sigma(t)} \sigma(m')$.*

Proof. Let $m \xrightarrow{t} m'$. First, we show that $\sigma(t)$ is enabled in $\sigma(m)$. Let $[p, \sigma(t)] \in F$. By symmetry, $[\sigma^{-1}(p), t] \in F$, too, and, since t is enabled at m, $m(\sigma^{-1}(p)) \geq W([\sigma^{-}1(p), t]) = W([p, \sigma(t)])$. Thus $\sigma(m)(p) = m(\sigma^{-1}(p)) \geq W([p, \sigma(t)]$ which shows that $\sigma(t)$ is enabled at $\sigma(m)$.

For the marking reached by firing $\sigma(t)$ in $\sigma(m)$, we have for all p, $\sigma(m)(p) - W([p, \sigma(t)] + W([\sigma(t), p]) = m(\sigma^{-1}(p)) - W([\sigma^{-1}(p), t]) + W(t, \sigma^{-1}(p)]) = m'(\sigma^{-1}(p)) = \sigma(m')(p)$. This concludes the "only if"direction. The "if"direction can be verified by applying the "only if"direction to σ^{-1}. ♢

Similar theorems exist for several other system description formalisms, and are usually not complicated to prove as long as all structural entities that influence the the system behavior are taken care of in the definition of symmetry.

The theorem can be extended to transition sequences, then saying that for each state reachable from m, an equivalent state is reachable from $\sigma(m)$. Thus, it makes sense not to explore $\sigma(m)$ if m has already been explored. This idea is captured by defining, for a symmetry group Σ, a relation $\equiv_\Sigma$ on markings where $m_1 \equiv_\Sigma m_2$ if and only if there is a $\sigma \in \Sigma$ such that $m_2 = \sigma(m_1)$. Since Σ is a group, $\equiv_\Sigma$ is an equivalence relation. By the above theorem, if one element of an equivalence class has a successor state in some other class, so do all elements in that class. This justifies the construction of a reduced state space where equivalence classes (represented by one element) are stored as nodes rather than states.

Definition 22 (Symmetrically reduced transition system) *Let $[S, E, A]$ be a transition system of a Petri net N and Σ a symmetry group*

for N. Then $[S_\Sigma, E_\Sigma, A']$, the symmetrically reduced transition system w.r.t. Σ, is defined by

- $S_\Sigma = \{[s]_\Sigma \mid s \in S\}$;
- *for all $s, s' \in A$ and $a \in A$, $[s, a, s'] \in E$ if and only if there is an $a' \in A'$ s.t. $[[s]_\Sigma, a', [s']_\Sigma] \in E$.*

Note that the definition of reduced graphs disregards actions.

Symmetrically reduced state spaces are bisimilar to their corresponding full state spaces with respect to atomic propositions that are not sensitive to symmetry. For those propositions, we define that an equivalence class satisfies an atomic proposition if and only if all its members do.

A state property ϕ is insensitive to a symmetry group Σ iff, for all $\sigma \in \Sigma$ and all states s, $s \models \phi$ implies $\sigma(s) \models \phi$. Many global properties (e.g., existence of deadlocks) are insensitive to all symmetries, all other properties can be forced to be insensitive by choosing an appropriate symmetry group Σ. If the largest symmetry group rendering a property insensitive consists solely of the identity, equivalence classes are all singletons, so no reduction can be achieved. Every larger symmetry group tends to condense the state space properly, by a factor depending of the number of symmetries in Σ.

Lemma 2 ([CEFJ96, ES96]; Class membership is bisimulation) *Let Σ be a symmetry group, $[S, E, A]$ a transition system, and $[S_\Sigma, E_\Sigma, A']$ a corresponding symmetrically reduced transition system w.r.t. Σ. Then the relation ρ with $s \rho s'$ iff $s' = [s]_\Sigma$ is a bisimulation w.r.t. all atomic properties that are insensitive to symmetry.*

Proof. Since atomic propositions are insensitive to symmetry, ρ relates only states that hold the same atomic propositions. Preservation of transition relations in both directions follows immediately from the main theorem for symmetries. ♢

Corollary 2 ([CEFJ96, ES96]; Symmetry preserves CTL*) *Every CTL* formula that contains only atomic propositions insensitive to symmetry is true of a transition system if and only if it is true of the corresponding symmetrically reduced transition system.*

6.2 Algorithms

There are two major tasks that need to be solved for an implementation of symmetry reduction. First, we need to determine a suitable symmetry group Σ that fits all needs established in the previous section. Second, we need procedures to construct a symmetrically reduced transition system based on that group Σ.

Here, we present an algorithm to compute a generating set of the Petri net symmetries that is actually used in the LoLA tool and seems to perform well. Despite its exponential worst case complexity, its run time grew only polynomially on all sequences of examples that we tested so far, except for nets explicitly constructed as counterexamples for the symmetry algorithm. The algorithm can be applied to arbitrary graph automorphism problems, but we have no experimental results concerning its behavior in domains other than Petri nets.

The algorithm is centered around a data structure that can be seen as *abstract automorphism*.

Definition 23 (abstract permutation) *Let $G = [V, E, c]$ be a labeled graph. A* constraint *is a pair $[V_1, V_2]$ with $V_1 \subseteq V$, $V_2 \subseteq V$. A permutation σ of V is* consistent *with $[V_1, V_2]$ ($\sigma \sim [V_1, V_2]$) iff $\sigma(V_1) = V_2$ (i.e. for all $v \in V_1$, $\sigma(v) \in V_2$ and for all $v \in V_2$, $\sigma^{-1}(v) \in V_1$). An* abstract permutation *is a set of constraints. A permutation σ is* consistent *with an abstract permutation α ($\sigma \sim \alpha$) iff σ is consistent with all constraints in α. With Π_α, we denote the set of permutations consistent with α.*

Examples. By definition, every graph automorphism is consistent with the abstract permutation $\alpha_0 = \{[V_d, V_d] \mid d \in D]\}$ where D is the color domain used for labeling the graph, and $V_d = \{v \mid v \in V \text{ and } c(v) = d\}$. In a Petri net $N = [P, T, F, W, m_0]$, every symmetry mapping some given marking m to another given marking m' is consistent with the abstract symmetry $\alpha_{m,m'} = \{[\{p \mid p \in P, m(p) = i\}, \{p \mid p \in P, m'(p) = i\}] \mid i \in \mathbf{N}\}$. A concrete permutation σ is the only permutation being consistent with $\alpha_\sigma = \{[\{v\}, \{\sigma(v)\}] \mid v \in V\}$. For $V = \{v_1, \ldots, v_n\}$, all graph automorphisms in the orbit O_{ij} are consistent with $\alpha_{ij} = \{[\{v_k\}, \{v_k\}] \mid 1 \leq k \leq i-1\} \cup \{[\{v_i\}, \{v_j\}]\}$.

Corollary 3 (Simple properties of abstract permutations)

- $\Pi_{\alpha_1 \cup \alpha_2} = \Pi_{\alpha_1} \cap \Pi_{\alpha_2}$;

- *If there is a* $[V_1, V_2] \in \alpha$ *and* $|V_1| \neq |V_2|$ *then* $\Pi_\alpha = \emptyset$*;*
- $\Pi_\alpha = \Pi_{\alpha \setminus \{[\emptyset,\emptyset],[V,V]\}}$*;*
- $\Pi_{\alpha \cup \{[V_1,V_2],[V_1',V_2']\}} = \Pi_{\alpha \cup \{[V_1 \cap V_1', V_2 \cap V_2'],[V_1 \setminus V_1', V_2 \setminus V_2'],[V_1' \setminus V_1, V_2' \setminus V_2]\}}$*;*
- $\Pi_{\alpha \cup \{[V_1,V_2]\}} = \Pi_{\alpha \cup \{[V_1,V_2],[V \setminus V_1, V \setminus V_2]\}}$.

By the example above, it is possible to specify orbits as abstract permutations. A collection of one graph automorphism per orbit forms a generating set, as pointed out in the previous section. Thus, we need a procedure to calculate one permutation that is consistent with a given abstract permutation and is an actual graph automorphism. We solve this problem by refining the abstract permutation until finally arriving at an abstract permutation like the α_σ in the above examples. The only permutation consistent with such an abstract permutation shall be a graph automorphism.

Our algorithm uses two transformations, REFINE and DEFINE. REFINE narrows the search space by replacing an abstract permutation by a more specific one without changing the set of consistent automorphisms. DEFINE partitions the search space. It replaces an abstract permutation by a set of abstract permutations such that each automorphism consistent with the old abstract permutation is consistent with exactly one of the new abstract permutation.

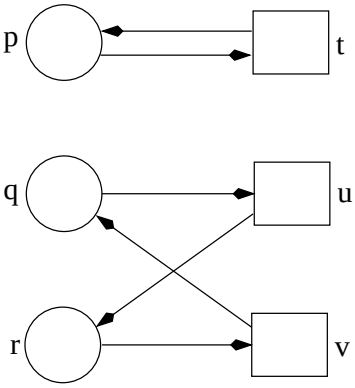

Figure 6.3: Running example for illustrating REFINE and DEFINE

REFINE is based on the fact that every graph automorphism must respect the edge relation. This preservation extends to abstract permutations in the following sense:

Lemma 3 (Abstract permutations and edge relation) *Let α be an abstract permutation, $\{[V_1, V_2], [V_1', V_2']\} \subseteq \alpha$, $c \in D$ a label, and σ a graph automorphism consistent with α. Then, for every $v \in V_1$, v has as many edges labeled c to (from, resp.) nodes in V_1' as $\sigma(v)$ (which is in V_2) has edges labeled c to (from, resp.) nodes in V_2'.*

Notice that we do not require $[V_1, V_2] \neq [V_1', V_2']$.

Proof. Let $v_1, \ldots, v_n$ be all the nodes in V_1' that have an edge labeled c originating at v (i.e., for all i $(1 \leq i \leq n)$, $[v, c, v_i] \in E$, $v_i = v_j$ implies $i = j$, and $[v, c, v'] \in E$ implies $v' = v_i$ for some i). Since $\sigma \sim \alpha$ and $\{[V_1, V_2], [V_1', V_2']\} \subseteq \alpha$, we have $\{\sigma(v_1), \ldots, \sigma(v_n)\} \subseteq V_2'$, and $\sigma(v_i) = \sigma(v_j)$ implies $i = j$, since σ is a bijection. Moreover, since σ is a graph automorphism, we have $[\sigma(v), c, \sigma(v_i)] \in E$, for all i $(1 \leq i \leq n)$.

Assume that there is a v' such that $v' \in V_2'$, $[\sigma(v), c, v'] \in E$, and $v' \neq \sigma(v_i)$, for all i $(1 \leq i \leq n)$. Then, since σ^{-1} is a graph automorphism as well, we have $[v, c, \sigma^{-1}(v')] \in E$ and, by definition of abstract permutations, $\sigma^{-1}(v') \in V_1'$. Since v' is different from all the $\sigma(v_i)$, $\sigma^{-1}(v')$ is different from all the v_i, which contradicts the assumption that $\{v_1, \ldots, v_n\}$ is exactly the set of nodes connected with v.

The claim for edges *to* v can be proven analogously. $\diamondsuit$

Corollary 4 (Narrowing constraints) *Let α be an abstract permutation, $c \in D$ a label, and σ a graph automorphism. Let $\{[V_1, V_2], [V_1', V_2']\} \subseteq \alpha$. $\sigma \sim \alpha$ if and only if $\sigma \sim \alpha \cup \{[V_1^{c,k}, V_2^{c,k}] \mid k \in \mathbf{N}\}$, where $V_1^{c,k}$ is the set of nodes in V_1 that have exactly k edges labeled c to nodes in V_1', and $V_2^{(k)}$ is the set of nodes in V_2 that have exactly k edges labeled c to nodes in V_2'.*

An equivalent corollary holds for edges *from* nodes in V_1. By the previous lemma, the new constraints do not establish any new restrictions to *automorphisms* while they do eliminate other permutations. Furthermore, after having added the new constraint, the original constraint $[V_1, V_2]$ becomes meaningless since, as can be verified easily, $\sigma \sim [V_a, V_b]$ and $\sigma \sim [V_c, V_d]$ implies $\sigma \sim [V_a \cup V_c, V_b \cup V_d]$.

These considerations justify the following REFINE transformation:

> *(REFINE transformation) Given an abstract permutation α, two constraints $[V_1, V_2]$ and $[V_1', V_2']$ in α (not necessarily different), and a label $c \in D$, replace $[V_1, V_2]$ by the set $\{[V_1^{c,k}, V_2^{c,k}] \mid$*

$k \in \mathbf{N}\}$ *of constraints where the* $V_1^{c,k}$ *and* $V_2^{(k)}$ *are defined as in Cor. 4.*

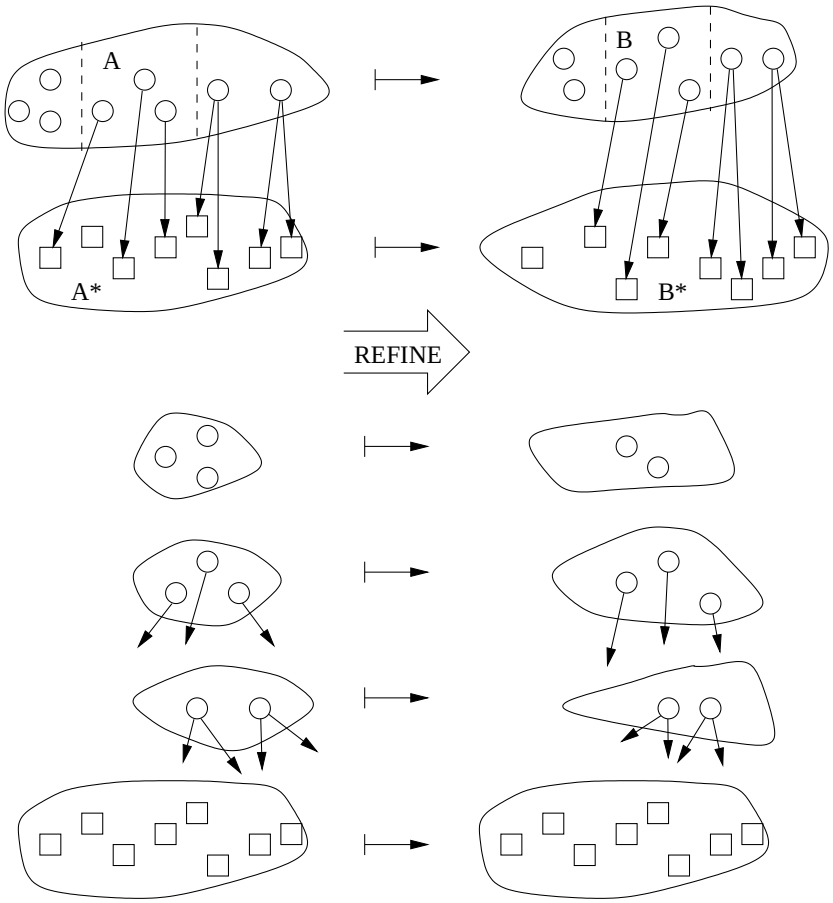

Figure 6.4: A refine step sketched for a Petri net setting. The nodes on the left can be different from the nodes on the right.

An equivalent operation can be applied for the reverse arc orientation. REFINE can be applied for arbitrary constraints in α, including the ones introduced by earlier application of REFINE.

Example. Consider the net in Fig. 6.3, and let

$$\alpha = \left\{ \begin{array}{c} [\{q\},\{r\}] \\ [\{p,r\},\{p,q\}] \\ [\{t,u,v\},\{t,u,v\}] \end{array} \right\}$$

First, consider the second constraint. Its left side has arcs to t (from p) and v (from r), but not to u. Its right side has arcs to t (from p) and u (from q). Thus, the transition constraint splits into $[\{t, v\}, \{t, u\}]$ and $[\{u\}, \{v\}]$. Using the same place constraint, but the inverse arc orientation, we have that the left side has arcs from t (to p) but not from v while the right side has arcs from t (to p) but not from u. Thus, $[\{t, v\}, \{t, u\}]$ can be split into $[\{t\}, \{t\}]$ and $[\{v\}, \{u\}]$. Using $[\{t\}, \{t\}]$ on $[\{p, r\}, \{p, q\}]$, splits the latter into $[\{p\}, \{p\}]$ and $[\{r\}, \{q\}]$. This way, we obtained an abstract permutation of singletons that represents the permutation $p \mapsto p, q \mapsto r, r \mapsto q, t \mapsto t, u \mapsto v, v \mapsto u$ which is in fact a symmetry of the net in Fig. 6.3.

Starting with constraints $[\{p\}, \{q\}]$ and $[\{t, u, v\}, \{t, u, v\}]$ leads to an inconsistency. Using arcs from the place constraint, the transition constraint splits into $[\{t\}, \{u\}]$ and $[\{u, v\}, \{t, v\}]$. Using the other arc orientation, the latter constraint is split into $[\{u, v\}, \{t\}]$ and $[\emptyset, \{v\}]$ since p has arc neither from u nor v while q does have an arc from v. Consequently, the considered net does not have a symmetry where p is mapped to q.

With REFINE*, we denote a repeated application of REFINE until, for all pairs of constraints, REFINE replaces a constraint only by the constraint itself and the constraint of empty sets. Our tool LoLA contains a polynomial time implementation of REFINE* for Petri nets.

It may occur that a REFINE transformation leads to a constraint where left and right sides have different cardinality. This situation indicates clearly that there is no permutation consistent with that constraint. Since the above lemma is nevertheless true, this indicates that there is no graph automorphism consistent with the original abstract permutation.

A DEFINE transformation takes a constraint $[V_1, V_2]$ of non-singletons, picks an element $v \in V_1$, and assigns it to all its possible images in V_2. That is, for each $v' \in V_2$, the constraint $[V_1, V_2]$ is replaced by the two constraints $[\{v\}, \{v'\}]$ and $[V_1 \setminus \{v\}, V_2 \setminus \{v'\}]$. Obviously, each graph automorphism σ that is consistent with $[V_1, V_2]$ is consistent with one of the new pairs of constraints—namely the one where $v' = \sigma(v)$.

> *(DEFINE transformation) Given an abstract permutation α, a constraint $[V_1, V_2]$ of non-singletons, and an element $v \in V_1$, replace α by the set of abstract permutations $\{(\alpha \setminus \{[V_1, V_2]\}) \cup \{[\{v\}, \{v'\}], [V_1 \setminus \{v\}, V_2 \setminus \{v'\}]\} \mid v' \in V_2\}$.*

Proposition 5 (Correctness of DEFINE) *If a permutation σ is consistent with an abstract permutation α, then it is consistent with exactly one of*

the abstract permutations that result of a DEFINE transformation on α.

Usually, the new constraints introduced by DEFINE enable further REFINE transformations. Thus, a symmetry calculation consists of alternating applications of REFINE* and DEFINE. Every sequence of applications ends in abstract permutations where left and right parts of a constraint have different cardinality (and therefore no automorphism is consistent), or in an abstract permutation where all constraints are pairs of singleton sets. Both REFINE and DEFINE preserve the presence of vertices. This means that if a vertex occurs on the left (right) side of some constraint in the original abstract permutation, it occurs as well on the left (right side) of every abstract permutation resulting from REFINE or DEFINE. Thus, starting from an abstract permutation that covers all vertices, any resulting abstract permutation consisting of singletons will completely define a permutation. Checking whether such a permutation is a graph automorphism is easy—in [Sch00a] we show that even the fact that a REFINE* does not further split an abstract permutation of singletons is already sufficient to qualify the corresponding actual permutation as graph automorphism.

The following algorithm summarizes our approach. It computes one graph automorphism consistent with a given abstract permutation α_0 if there is one, otherwise it returns $\perp$. We assume that there is a procedure REFINE* that takes an abstract permutation as input and returns the result of the REFINE* transformation (another abstract permutation. We assume further that DEFINE takes an abstract permutation and returns the corresponding result of the DEFINE transformation (a set of abstract permutations). Then, Singletons checks whether all constraints in α consist of singletons, and IsAutomorphism checks whether the unique permutation consistent with such an abstract permutation is a graph automorphism. Finally, IsConsistent returns true if all constraints in α have equal cardinalities in both sides. The algorithmic ideas behind all these subroutines should be clear from our considerations.

In general, the sketched procedure for finding one graph automorphism that is consistent with a given abstract permutation has exponential time complexity. The reason is that DEFINE replaces an abstract permutation by a set of abstract permutations each of which must be checked for consistent graph automorphisms. A significantly better worst case complexity cannot be expected since the decision problem of graph isomorphism can be reduced to the above procedure. Given two graphs $[V_1, E_1]$ and $[V_2, E_2]$, the graph

Figure 6.5: Computing an automorphism from an abstract specification

```
1   var A: set of abstract permutations
2   var α: abstract permutation
3   function CompAut(α0: abstract permutation) : permutation
4   begin
5       A := {α0};
6       while A ≠ ∅ do
7           choose α ∈ A; A := A \ {α};
8           α := REFINE*(α);
9           if IsConsistent(α) then
10              if Singletons(α) then
11                  if IsAutomorphism(α) then
12                      return α; exit;
13                  end;
14              else
15                      A := A∪ DEFINE(α);
16              fi
17          fi
18      done
19      return ⊥;
20  end.
```

isomorphism problem asks whether there is a bijection between the vertices of both graphs that respects edges (in the same sense as automorphisms). Graph isomorphism is a problem that is known to be in NP while neither NP completeness nor polynomiality of the problem could be proven so far, despite considerable efforts. Assuming that both graphs have disjoint vertex sets, the problem can be easily reduced to the above procedure by asking whether the graph $[V_1 \cup V_2, E_1 \cup E_2]$ has an automorphism that is consistent with the abstract permutation $\{[V_1, V_2], [V_2, V_1]\}$.

However, in most examples we tested it turned out that either the initial REFINE* yields an inconsistent abstract permutation, or each abstract permutation resulting from the subsequent DEFINE had a consistent graph automorphism. In such cases (using a last in first out strategy for **choose** in line 7) the whole procedure CompAut finds a graph automorphism (or

detects that there is none) in polynomial time. The reason is that every REFINE* runs in polynomial time (as claimed earlier), and DEFINE runs in constant time and can only be applied polynomially often (it always adds another singleton constraint).

In the previous section we have seen that for each graph there are polynomially many orbits, each orbit can be characterized by an abstract permutation, and each nonempty orbit contributes exactly one (arbitrary) element to a generating set. Thus, we need a polynomial number of calls to CompAut in order to compute a generating set for the full automorphism group of a given graph. Subgroups of the automorphism group can be computed easily by adding appropriate constraints to the orbit specifications.

Example. Consider the net in Fig. 6.3. In order to find a generating set of the symmetries, we need to investigate the following abstract permutations:

$$[\{p\},\{p\}],[\{q,r\},\{q,r\}],[\{t,u,v\},\{t,u,v\}]$$
$$[\{p\},\{q\}],[\{q,r\},\{p,r\}],[\{t,u,v\},\{t,u,v\}]$$
$$[\{p\},\{r\}],[\{q,r\},\{p,q\}],[\{t,u,v\},\{t,u,v\}]$$
$$[\{p\},\{p\}],[\{q\},\{q\}],[\{r\},\{r\}],[\{t,u,v\},\{t,u,v\}]$$
$$[\{p\},\{p\}],[\{q\},\{r\}],[\{r\},\{q\}],[\{t,u,v\},\{t,u,v\}]$$

We do not need to continue this list with transition constraints since it can be shown that in a graph automorphism, every mapping on the places determines uniquely the corresponding mapping on the transitions (unless there are parallel transitions, i.e. transitions identically connected to places). The fifth abstract permutation leads, as seen earlier, to a symmetry, the second and third ones to an inconsistency. The fourth abstract permutation leads to the identity through REFINE*. REFINE* on the first abstract permutation splits the transition constraint into $[\{t\},\{t\}]$ and $[\{u,v\},\{u,v\}]$. Then, a DEFINE is necessary to reach either the identity or the mentioned symmetry, depending on which branch of DEFINE is taken first.

We turn now to the problem of computing a symmetrically reduced transition system. Theoretically, states of the reduced system correspond to equivalence classes of states of the original system. In practice, these equivalence classes are represented by one of their elements. This is an appropriate representation since, as pointed out in the theory section, every successor class of an equivalence class c contains a successor of any member of c. In order to adapt the existing state space exploration routines, we need to solve the following problem which we refer to as the "symmetry integration problem":

Given a symmetry group Σ, a state s, and a set of states S, decide whether there is a $\sigma \in \Sigma$ such that $\sigma(s) \in S$.

In this setting, S is the set of already known states (representatives of already explored equivalence classes), and s is the current state, for which we must decide whether its class is already represented in S. Σ may be given as its generating set, or just by its specification (for instance, as abstract permutation) without having any pre–computed information. We compare four techniques for solving the above problem. Two of the four techniques rely on an available generating set as presented in the theory section. The other two techniques do not require any preprocessed generating set. Two of the four techniques solve the above problem accurately while the remaining two techniques solve it only approximately. An approximate solution to the symmetry integration problem is a procedure which answers "no"whenever there is no $\sigma \in \Sigma$ such that $\sigma(s) \in S$ but may answer "yes"or "no"when there is a $\sigma \in \Sigma$ where $\sigma(s) \in S$. The consequence of using an approximate solution to the symmetry integration problem in a state space exploration procedure is that, for some equivalence classes, more than one representative may appear in the reduced state space. This situation does not threaten correctness of verification results but yields larger state spaces. In return, approximate procedures may be much more time efficient than accurate procedures. In fact, our approximate procedures have polynomial run time while Junttila was able to show [Jun01] that an accurate solution to the symmetry integration problem is at least as hard as the graph isomorphism problem. He showed, among others, that deciding equivalence of states of a Petri net is equivalent to the graph isomorphism problem. Thereby, the complexity is the same, regardless of the presence or absence of a preprocessed generating set of the automorphism group.

The first solution to the integration problem is called *iteration of symmetries*. It originated on an earlier implementation of symmetry reduction in the Petri net tool INA [RS98]. There, not only a generating set, but the complete symmetry group involved was calculated in preprocessing, and explicitly stored as a linked list. The integration problem was solved as a traversal of the list of symmetries, checking for each symmetry in the list whether $\sigma(s) \in S$. Of course, this approach worked only for small systems and sparse symmetry groups. The solution that we refer to as iteration of symmetries has been developed from the INA procedure in two steps. First, we replaced the explicit list of symmetries by the generating set introduced

above and implemented the traversal of all symmetries using the capability of our generating set to enumerate the full set of symmetries without repetition (see previous section for details). Then we observed that, assuming a decision tree as representation of S, it was possible to skip many symmetries in the enumeration process by taking into consideration the depth at which a test $\sigma(s) \in S$ failed in the decision tree. The number of symmetries skipped turned out to be large enough to more than compensate the run time overhead that was introduced by composing symmetries from the generating set rather than just moving to the next item in the explicitly stored list of symmetries (not to mention the gains in memory efficiency that result from storing only generators permanently rather than all symmetries).

In a decision tree, every nonterminal layer corresponds to a state variable, and every offspring edge to a particular value of that variable. A state corresponds to a path through the decision tree starting at the root node, and assuming, at every layer the value associated with the chosen edge as the value of the corresponding state variable. A decision tree stores a set of all those states that correspond to a path in the decision tree from the root down to the terminal layer.

Checking whether a state is element of the set represented by a decision tree amounts to traversing the tree from the root to the terminal layer, thereby always proceeding through the edge corresponding to the value of the state variable in the current state. If, at some layer i, there is no edge associated to the value of the current state, the state is not in the set represented by the tree, otherwise it is. In case that a state is not in the tree, we say that $i + 1$ is the *failing layer* for that state. For example, state $[1, 1, 5]$ is not in the set represented in Fig. 3.3 on page 41, and would have 2 as its failing layer.

The failing layer k for a state s provides valuable information, namely that every other state s' that is equal to s on the first k components cannot be in the set represented by the decision tree either. It is this information that we use to skip symmetries in the enumeration mentioned above. Let m be a marking of a Petri net and σ_1, σ_2 symmetries such that $\sigma_1(p_i) = \sigma_2(p_i)$ for $1 \leq i \leq k$ (the first k places of the Petri net, assuming the order they are layered in the decision tree). Then, it holds for those p_i $(1 \leq i \leq k)$:

$$\sigma_1^{-1}(m)[p_i] = m(\sigma_1(p_i)) = m(\sigma_2(p_i)) = \sigma_2^{-1}(m)[p_i].$$

Hence, if two symmetries are equal on some places, then their inverse application to some state yields equal values on those places.

When enumerating all symmetries using the standard generating set, consecutive symmetries have naturally equal values on the first few places. The reason is that every symmetry is the product of n generators $\sigma_{1j_1} \circ \cdots \circ \sigma_{nj_n}$ where σ_{ij} is an element of the orbit O_{ij}. That is, σ_{ij} is the identity on at least the first i elements. This means that every composition $\sigma_{1j_1} \circ \cdots \circ \sigma_{kj_k} \circ \sigma_{k+1j'_{k+1}} \circ \cdots \circ \sigma_{nk'_n}$ has the same values on the first k graph vertices as $\sigma_{1j_1} \circ \cdots \circ \sigma_{nj_n}$.

Combining the observation on the structure of our generating set with the previous observation on failing layers of the decision tree, we can conclude: if, for some $\sigma = \sigma_{1j_1} \circ \cdots \circ \sigma_{nj_n}$, it turns out that $\sigma^{-1}(m)$ is not in the set represented by the given decision tree and fails at layer k, all σ' generated as $\sigma' = \sigma_{1j_1} \circ \cdots \circ \sigma_{kj_k} \circ \sigma_{k+1j'_{k+1}} \circ \cdots \circ \sigma_{nk'_n}$ can be skipped since σ' has the same first k values as σ, thus $\sigma'^{-1}(m)$ would have the same first k components as $\sigma^{-1}(m)$, and consequently $\sigma'^{-1}(m)$ would fail at the same layer k as $\sigma^{-1}(m)$. Note that in our setting, we would check all $\sigma^{-1}(m)$ for containment in S instead of $\sigma(m)$. This is no problem since Σ is a group, so all elements of Σ are the inverse of some other element of Σ. Fig. 6.6 sketches the algorithmic idea of iterating symmetries with skipping. It uses a routine $\text{Next}(i, j)$ that returns the smallest index k such that $k > j$ and the orbit O_{ij} is not empty (i.e., there is a symmetry $\sigma_{ij} \in O_{ij}$ in the generating set). If no such k exists, the routine returns $n + 1$ where n is the number of nodes in the Petri net. Function IterateSymm return true if there is a $\sigma \in \Sigma$ such that $\sigma(m) \in S$, and false otherwise.

Though the skipping of symmetries during the enumeration process speeds up symmetry integration significantly, it still behaves weak for dense symmetry groups. When such groups are involved, there are still too many containment checks that need to be performed. This is unfortunate since dense symmetry groups have the potential of better reduction (more states can be equivalent). The second solution to the symmetry integration problem does not depend on a preprocessed generating set which is an advantage for dense symmetry groups since their generating set is larger than generating sets of sparse symmetry groups. The idea of the second solution is to iterate states in S rather than symmetries. For each state m' in S, the question of whether there is a symmetry that maps m to m' is equivalent to the question whether there is an automorphism that is consistent with the abstract permutation $\alpha_{m,m'}$ (see example on page 118) and can be solved using procedure CompAut.

Figure 6.6: Integrating symmetries by iterating symmetries

```
1  var S: set of states (as decision tree)
2  function IterateSymm(m: marking) : boolean
3  var j_1, ..., j_n, fl: integer
4  begin
5     j_1 := 1;
6     while j_1 ≤ n do
7        j_2 := 2;
8        while j_2 ≤ n do
9           ...
10                      j_{n-2} := n − 2;
11                      while j_{n-2} ≤ n − 2 do
12                         j_{n-1} := n − 1;
13                         while j_{n-1} ≤ n − 1 do
14                            if (σ_{1j_1} ∘ ... σ_{n-1 j_{n-1}})^{-1}(m) ∈ S then
15                               return true;
16                            end
17                            fl := failing layer of prev. search;
18                            j_{n-1} := next(n − 1, j_{n-1});
19                            if fl < n − 1 then j_{n-1} := n + 1; end;
20                         end;
21                         j_{n-2} := next(n − 2, j_{n-2});
22                         if fl < n − 2 then j_{n-2} := n + 1; end;
23                      end;
24          ...
25          j_2 := next(2, j_2);
26          if fl < 2 then j_2 := n + 1; end;
27       end;
28       j_1 := next(1, j_1);
29    end;
30    return false;
31 end.
```

Of course, iterating all states in S is too expensive. However, if we have a hash function that respects symmetry (i.e. equivalent states have equal hash values), and S is partitioned according to that hash function, it is sufficient to iterate only states in the hash class of the current state. This way, the number of calls to CompAut can be kept reasonably small, especially for dense symmetry groups.

Applying this technique to sparse groups leads to bad results. Besides the fact that IterateSymmetries works much more efficient for sparse symmetry groups, we observed that symmetry respecting hash functions tend to work better for dense symmetry groups. With a dense symmetry group, more states sharing the same hash value are equivalent than with a sparse group. Thus, hash classes of the reduced state space of a system with a dense symmetry group contain less members than comparable state spaces of systems with sparse symmetry groups, causing less calls to CompAut. Fig. 6.7 depicts the second solution to the integration problem.

Figure 6.7: Integrating symmetries by iterating states

```
1     var S: set of states
2     var h : S ⟶ N: hash function
3     function IterateStates(m: marking) : boolean
4     begin
5         for all m' such that m' ∈ S and h(m') = h(m) do
6             if CompAut(α_{m,m'}) ≠ ⊥ then
7                 return true;
8             end
9         end;
10        return false;
11    end.
```

In the first two approaches to the integration problem, any member of an equivalence class can be the representative of its class in the set of states S. The remaining two approaches are based on the idea to have a distinguished member of every equivalence class in S, the so-called *canonical representative*. In our approaches, the canonical representative shall be the lexicographically

smallest member of its equivalence class. The core of these approaches is a procedure to transform a state into the canonical representative of its equivalence class. This transformation is immediately appended to successor state computation. Using such a procedure, the symmetry integration problem reduces to plain search in S.

The first procedure to canonicalize states—the third approach to the symmetry integration problem—relies on the special structure of the generating set used throughout this chapter. We observe that generators that become members of the generating set as elements of an orbit O_{ij} are the identity on the first $i-1$ vertices. Thus, they do not alter the value of the first $i-1$ state components. So, we can apply generators coming from O_{ij} with small i to move values as small as possible to the first components of a state, and then use generators coming from O_{ij} with larger i to put small values on the remaining components, without altering the values on the first (lexicographically more significant) components. Our implementation of this algorithmic idea is sketched in Fig. 6.8 (let n be the number of nodes of the Petri net and next(i,j) as in Fig. 6.6).

Figure 6.8: Integrating symmetries by canonicalizing states using generators

```
1     var S: set of states
2     function CanRepGenerators(m: marking) : marking
3     var i, j: integer;
4     begin
5         for i := 1 to n do
6             j := i;
7             while j ≤ n do
8                 if σ_ij^-1(m) < m then
9                     m := σ_ij^-1(m);
10                    j := next(i, j);
11                end
12            end
13        end;
14        return m;
15    end.
```

The procedure has obviously polynomial run time, so it can only be an approximate approach (since graph isomorphism can be rephrased as equivalence of two Petri net markings [Jun01], and equivalence of markings can be implemented by canonicalizing both and comparing the results). The reason is that generators in the early iterations of the for loop may permute *all* the remaining components so that some value can be put to a position where it cannot be moved away in later iterations. The problem does not occur, among others, for a full permutation symmetry group where our canonicalization procedure implements a sort, and for ring style symmetry groups. In the latter kind of groups, all generators come from orbits with the same first index, so the canonicalization implements an exhaustive examination of all symmetries. This may be one of the reasons why data type related symmetry approaches handle only permutation and ring groups (they do all integrate symmetries by some canonicalization procedure based on sorting).

Instead of extending the present approximate solution to an accurate canonicalization procedure, we decided to use the approximate solution as it is, thus providing an approximate solution to the symmetry integration problem. The procedure turns out to be highly efficient, but can lead, of course, to larger state spaces than the previous two methods.

The last of the four methods is yet another canonicalization procedure. It was proposed by Junttila [Jun00]. In contrast to the previous method it does not rely on a preprocessed generating set. On the other hand, it is significantly slower. Thus, its usefulness is restricted to those cases where the computation of the generating set itself is the time or space bottleneck of the verification, and the iteration of states method does not perform well. We are not going to present this method in detail. It basically computes a symmetry as in CompAut, but uses a strategy in choose (line 7) to proceed with subproblems first that lead to a symmetry which transforms m into its canonical representative. For an accurate solution, backtracking to other subproblems is necessary. Since the accurate solution turned out to be too slow, we limited the number of backtracking thus having another approximate solution to the canonicalization problem.

6.3 Performance

The first experiments concern the run time of our algorithm to compute the generating set of a symmetry group of a Petri net. For this purpose, we run

LoLA two times on each net. First, we let LoLA read a net, set up all internal data structures, compute the generating set, and terminate without generating the reduced state space. Second, we let LoLA just read the net, set up the data structures, and terminate. The difference between these two end-to-end run times is reported below as the preprocessing overhead for computing the generating set of the symmetry group. We reported additionally the number of graph automorphisms and the number of generators explicitly stored in LoLA.

The first table contains data for the philosophers system, having a ring like structure, and the readers/writers access to a database, having a full permutation style symmetry group. For ring style groups, all symmetries except identity are generators. For dense groups, the full set of symmetries could not be stored explicitly already for trivial problem size.

Table 6.2: Symmetry preprocessing I

n		PH n			DA n	
	# symm	# gen	time	#symm	#gen	time
5	5	4	0.001	14400	20	0.007
10	10	9	0.013	$(10!)^2$	90	0.065
50	50	49	0.202	$(50!)^2$	2450	7.072
51	51	50	0.207	$(51!)^2$	2550	7.624
100	100	99	0.733	$(100!)^2$	9900	83.05
200	200	199	1.993	$(200!)^2$	39800	1099.136
500	500	499	20.748			
1000	1000	999	103.805			
1001	1001	1000	67.522			
2000	2000	1999	564.394			
2001	2001	2000	375.442			

The nonmonotonic run time behavior, for instance from PH 1000 to PH 1001 can be explained as follows. Whenever LoLA finds a generator, it iteratively composes that generator with itself. Since orbits are, in general, no subgroups, this composition may be an element of another orbit. In such a case, LoLA uses the composed element and does not dive into the calculation concerning that orbit. Composing symmetries is obviously much cheaper than the REFINE*/DEFINE procedure. Now, a symmetry, iteratively composed with itself, forms a subgroup of the original symmetry group. As a

subgroup, its number of elements divides the size of the original group. Now, since 1000 has many denominators, it can happen that the subgroup spanned by the found symmetry is proper, so only some orbits are covered with multiples of the found symmetry. For the remaining orbits, REFINE*/DEFINE must be applied. If the size of the complete symmetry has only few denominators, as for PH 1001, it is most likely that the subgroup spanned by a found symmetry is equal to the full group, thus covering all orbits. No further descent into REFINE*/DEFINE applies.

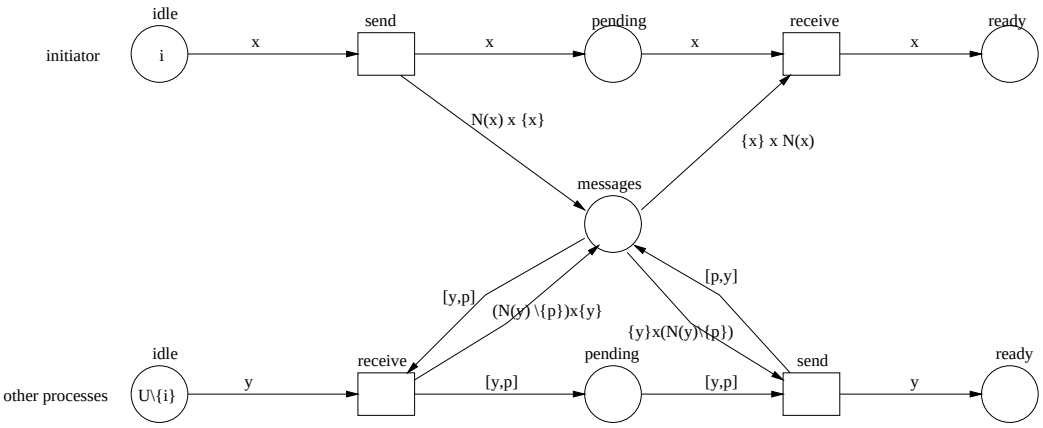

Figure 6.9: ECHO: an algorithm for propagation of information with feedback. i is the initiating process, U the set of all processes, and N the neighborhood relation that depends on the connection network. Messages consist of a receiver and a sender, in this order. The actual content of messages is abstracted.

The next table shows the behavior of our symmetry algorithm to grid style networks. We have as parameters the number of dimensions (1=line, 2=square, 3=cube, and so on), and as second parameter the number of vertices per line. Thus, a grid with dimension d and n vertices per line has a total of n^d vertices. In ECHO, each vertex corresponds to an agent, each edge to a message passing channel. The algorithm implements a propagation of information with feedback. In this algorithm, there is a distinguished initiator process. We put the initiator into the center of the grid, so we consider only odd values for n. In SIMPLE, vertices correspond to agents that request access to some resource in mutual exclusion from all neighbors in the grid. For every edge in the grid, there is a semaphore granting mutual exclusion.

Table 6.3: Symmetry preprocessing II

d /n	# symm	# gen	ECHO d/n time	SIMPLE d/n time
2/3	8	4	0.021	0.007
2/7	8	4	0.434	0.048
2/11	8	4	2.852	0.268
2/15	8	4	7.338	0.893
2/17	8	4		1.449
2/30	8	4		17.091
2/50	8	4		104.222
3/3	48	10	0.499	0.037
3/5	48	10	18.768	0.762
3/8	48	10		9.716
4/3	384	21	13.018	0.227
4/5	384	21		24.158
4/7	384	21		326.254
5/3	3840	41		4.244
6/3	46080	78		46.806

In this example, we can see that an increasing size of the problem with constant size of the symmetry group (constant d and growing n) has much less impact on run time than a more complex symmetry group. We interpret this as follows: For equal system size, a more heterogeneously (asymmetrically) structured system can be handled easier than a more regularly structured system. We have therefore only little doubts that our algorithms behave on academic examples similarly to more heterogenous real-world examples.

The remaining tables illustrate the condensation of state spaces using symmetries in isolation. We compare different symmetry integration methods.

Table 6.4: Reduced graph generation by iterating symmetries: PH n

n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
10	59048	393650	1.636	5933	39550	0.732
12	531440	4251516	24.201	44367	354932	10.274
13				122642	1062893	40.616
14				341801	3190138	152.573

Table 6.5: Reduced graph generation by iterating symmetries: DA n

n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
5	217	770	0.032	14	82	0.030
8	2368	12416	0.082	20	190	3.262
9	5201	30114	0.152	22	236	0.113
10	11364	71880	0.354	24		> 1000
11	24697	169202	0.824	26		> 1000

Table 6.6: Reduced graph generation by iterating symmetries: SIMPLE d/n

d/n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
2/2	7	16	0.024	3	8	0.028
2/3	63	304	0.039	20	100	0.033
2/5	55447	688478	2.968	8369	103413	1.344
3/2	35	144	0.028	6	27	0.030
3/3	70633	897594	4.080	2294	29382	1.602
4/2	743	5664	0.059	21	172	0.079
5/2	254475	3689792	20.083	297	4357	10.672

Our results suggest the following conclusions. First, canonicalization based on the preprocessed generating set outperforms the other techniques concerning run time. It is more insensitive to the shape of the involved symmetry group. Its only disadvantage is that it can yield a substantially larger state space (particularly on grid examples). Nevertheless, canonicalization should be the default integration technique. The remaining techniques are more or less backups for the case that canonicalization based on the generating set fails. Depending on the reason for failure, we have different options. If computing the generating set worked well but state space generation ran out of memory, we can switch to iteration of symmetries for sparse symmetry groups, or to iteration of states. These two techniques are accurate thus producing potentially smaller state spaces. If the problem was already the calculation of the generating set, we can switch to either of the two methods that do not rely on the generating set. Thereby, iteration of states should only be applied to dense symmetry groups, and canonicalization without pre-processed generating set only if everything else fails.

Table 6.7: Reduced graph generation by iterating symmetries: ECHO d/n

d/n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
2/3	2628	9994	0.107	402	1556	0.101

Table 6.8: Reduced graph generation by iterating states: PH n

n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
5	242	805	0.023	50	165	0.111
10	59048	393650	1.636	5933	39550	356.037

Table 6.9: Reduced graph generation by iterating states: DA n

n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
10	11364	71880	0.354	24	287	1.040
11	24697	169202	0.824	26	343	1.535
20				44	1072	16.575
40				84	4142	287.282

6.4 Compatibility

Compatibility between the symmetry and the stubborn set methods has been studied for data type related symmetries of high level Petri nets in [Val91a], with the result that basic stubborn can be applied in connection with symmetries. This result relies on two observations.

First, implementations of stubborn sets and symmetries do not interfere with each other. While stubborn sets concern the set of actions to be explored in a state, symmetries concern the states to be stored and further explored. Thus, a combined application of both methods consists simply of computing, for a given state, a stubborn set as usual, exploring the enabled transitions in the stubborn set, and trying to find a symmetric image of the resulting states in the set of computed states.

Second, if a set U is a basic stubborn set in a state s, then $\sigma(U) = \{\sigma(u) \mid u \in U\}$ is basic stubborn in $\sigma(s)$ (since the requirements for basic stubborn sets concern the existence of certain executable sequences which

Table 6.10: Reduced graph generation by iterating states: SIMPLE d/n

d/n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
2/2	7	16	0.024	3	8	0.023
2/3	63	304	0.039	20	100	0.068
2/5	55447	688478	2.968	8369	103413	257.239
3/2	35	144	0.028	6	27	0.048
3/3	70633	897594	4.080	2294	29382	111.165
4/2	743	5664	0.059	21	172	0.048
5/2	254475	3689792	20.083	297	4357	25.448

Table 6.11: Reduced graph generation by iterating states: ECHO d/n

d/n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
2/3	2628	9994	0.107	402	1556	1.775

Table 6.12: Reduced graph generation by canonicalizing states: PH n

n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
10	59048	393650	1.636	5933	39550	0.327
12	531440	4251516	24.201	44367	354932	3.063
13				122642	1062893	10.071
14				341801	3190138	34.491

are insensitive to symmetry by the main theorem on symmetries). Thus, exploration of transitions in a stubborn set in s represents the exploration of transitions in a stubborn set in $\sigma(s)$.

The arguments in [Val91a] can be immediately applied to automorphism based Petri nets, to other models for system description, and to all other simple stubborn set methods (those that do not rely on the detection of strongly connected components).

Compatibility between symmetries and advanced stubborn set methods is a more involved problem. Every (terminal) strongly connected component C in a transition system without symmetry reduction corresponds to some (terminal) strongly connected component C' in the corresponding transition system with symmetry reduction as follows: for every state s' in C', C contains at least one state s that is equivalent to s', and for every state s in C, C' contains at least one state s' equivalent to s. However, one and the same C' may correspond to one or more C, and the equivalences between states in C and C' are not necessarily realized by the same symmetry.

However, it is possible to adapt advanced stubborn set methods such as the removal of ignored actions to the symmetrically reduced case. Compati-

Table 6.13: Reduced graph generation by canonicalizing states: DA n

n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
10	11364	71880	0.354	32	295	0.096
11	24697	169202	0.824	35	352	0.121
20				62	1090	0.831
40				122	4180	19.522
80				242	16360	546.773

Table 6.14: Reduced graph generation by canonicalizing states: SIMPLE d/n

d/n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
2/2	7	16	0.024	3	8	0.023
2/3	63	304	0.039	20	100	0.031
2/5	55447	688478	2.968	12800	158835	1.073
3/2	35	144	0.028	6	27	0.030
3/3	70633	897594	4.080	7503	96199	0.970
4/2	743	5664	0.059	29	234	0.067
5/2	254475	3689792	20.083	3062	44983	1.067

Table 6.15: Reduced graph generation by canonicalizing states: ECHO d/n

d/n	states (full)	edges(full)	time (full)	states (red.)	edges (red.)	time (red.)
2/3	2628	9994	0.107	595	2263	0.096

bility between symmetries and stubborn set methods with removal of ignored transitions means computing a symmetrically reduced transition system such that, by unfolding that transition system, a stubborn set reduced transition system without ignored actions is obtained. With the unfolded transition system we mean a transition system that contains all symmetric images of states in the symmetrically reduced transition system, with edges corresponding to symmetric images of edges in the reduced system.

For avoiding ignored actions in a symmetrically reduced transition system, we propose the following strategy. For every terminal strongly connected component C of the symmetrically reduced system, and every equivalence class of $[t]_{\equiv_\Sigma}$ of transitions w.r.t. the used symmetry group Σ, check whether for all states in C some element of $[t]_{\equiv_\Sigma}$ is enabled. If so, extend the stubborn set used in the root node of that component, by a stubborn superset of $[t]_{\equiv_\Sigma}$. By the correlation between components in the symmetrically reduced and unreduced systems it is easy to see that this strategy avoids ignored transitions in the unfolded transition system. For other requirements of advanced stubborn set methods, similar approaches should help.

There are no compatibility problems concerning the use of symmetries with breadth first or depth first search. For distributed search, we observe some restriction. When integrating symmetries through iterating symmetries, we need to check for presence of states $\sigma(s)$, for several symmetries s. Since these states can be distributed over several machines, we obtain a

large communication overhead. Thus, it is undesirable to use that integration method in connection with distributed search.

The same problem occurs for the integration of symmetries through iterating states. States of a hash class, that need to be iterated in this method, can be distributed over several processes, too. A distribution scheme that collects all states with one and the same hash value on one machine poses an undesirable restriction to possible load balancing procedures.

At least one integration technique—canonicalization—can be applied straightforward in connection with distributed search. We can construct a canonical representative of a state immediately after its generation. Then, we can ship the representative to the machine responsible for it. This method does not require more communication than state space generation without symmetries. The additional run time for computing the canonical representative reduces the frequency of communication events which is a desirable side effect.

6.5 Discussion

The symmetry method must be applied with care. First, in difference to the stubborn set method, the symmetry method, in most instances, requires preprocessing that can potentially be costly. Second, the overhead required for symmetrically reduced state space generation is more significant. Third, achievable reduction depends on the shape of the involved symmetry group. Fourth, there are different methods to integrate symmetries into state space generation which resolve the inherent space/time tradeoff differently.

Although our integration technique of canonicalizing states with the help of a pre-computed generating set for the symmetry group may produce larger reduced state spaces than other methods, it seems to be the most robust technique in many respects: it is constantly among the most time efficient techniques (independently of the shape of the symmetry group), and it causes the least compatibility problems with other state space reduction techniques or search strategies. It is thus the technique that can be used with the least expert knowledge. Since it is compatible with distributed search, its space inefficiency can be compensated with additional hardware investions. For a more experienced user, the other techniques offer valuable alternatives, should canonicalization fail.

The biggest advantage of the symmetry technique is that symmetrically

reduced systems are closely tied (bisimilar) to the original transition system. The reduction preserves therefore a large range of properties. This way, applying symmetries in connection with another reduction method does not constrain the other method's capabilities of property preservation.

Symmetries can be applied to virtually all system description formalisms. All one needs to do is to identify all features in the system description that influence the change of states, and to define a mapping that preserves these structural features. Many description formalisms rely on variables and data type operations so that the data type based symmetry approach can be applied. For the graph automorphism approach, we believe that there is an even broader application area, given that graphs are a ubiquitous concept in computer science.

The main drawback of the symmetry method is that a "perfect"symmetric structure of the system is required. Our method fails completely on systems that are constructed from a number of systems that are slight variations of each other.

We presented the graph automorphism based symmetry approach as an alternative to the data type based approach to symmetry. While the graph automorphism approach is more flexible concerning the shape of applicable symmetry groups, data type based symmetries require less time to determine symmetries, particularly for large systems. Thus, future research on symmetries should deal with the problem of how to bring together the two approaches to symmetry reduction and to combine their advantages.

Chapter 7

Coverability analysis

Coverability graph construction [KM69, Fin90] was one of the first verification algorithms for Petri nets. It provides a finite abstract representation of the state space of an unbounded (infinite state) Petri net. The core feature of the construction is a technique that later, in the framework of abstract interpretation [CC77], had been called *widening*. During a naive construction of an (abstract) state space, there are often situations where infinite sequences of (abstract) states would be constructed iteratively. A widening operation replaces early members of such a sequence by a larger (more abstract) element of the abstract space such that, after finitely many applications of widening, the original infinite sequence is completely subsumed by a more abstract but finite sequence. In the case of unbounded Petri nets, there are reachable states m, m' where $m \xrightarrow{*} m'$ and $m > m'$ (i.e., m' is greater than m in some (at least one) components and equal to m in the remaining components). Such a situation causes an infinite sequence of states, since the sequence w that transformed m into m' is executable at m' again, due to the monotonicity of the Petri net enabling rule. Thus, without widening we would end up with a strictly increasing sequence of states $m_0 = m \xrightarrow{w} m_1 = m' \xrightarrow{w} m_2 \xrightarrow{w} ....$ The widening operation used for coverability graph construction replaces all components of m' where it is greater than m by $\mathbf{N}$, i.e. shifts m' to an abstract state that represents all concrete states that are equal to m' where m' is equal to m, and can take any value where m' is greater than m. This way, the occurrence of w at the abstracted m' does not lead to a new state but back to the abstraction of m'. The widening operates the same way when m and m' are already abstract states.

7.1 Theory

We start with two implementation-independent definitions for coverability graphs. The first is a strict definition and is satisfied by the construction in [KM69], as well as by the original state space, but not by the minimal coverability graph in [Fin90]. The second, sloppy definition covers all three constructions. The reason for having the strict definition is that we shall provide results on preservation of properties that hold for the construction by Karp and Miller, but not for Finkel's definition of coverability graphs. In Sec. 7.2, we shall see that the respective constructions do indeed satisfy our definitions. Throughout this chapter, let ω be a symbol representing "infinity"added to the natural numbers. For an arbitrary actual natural number n, let $n + \omega = \omega$, $\omega - n = \omega$, and $n < \omega$. For a vector $\mu : P \longrightarrow \mathbf{N} \cup \{\omega\}$, let $\Omega\mu = \{p \mid \mu(p) = \omega\}$. For a transition t of a Petri net $N = [P, T, F, W, m_0]$, let Δt be a P-indexed vector holding $\Delta t[p] = W([t, p]) - W([p, t])$. For a transition sequence $w = t_1 t_2 \dots t_n$, let $\Delta w = \sum_{i=1}^{n} \Delta t_i$. Observe that $m \xrightarrow{w} m'$ implies $m' = m + \Delta w$ for arbitrary Petri net markings m, m'.

Definition 24 (Strict coverability graph) *Let $N = [P, T, F, W, m_0]$ be a Petri net. A transition system $[S_C, E_C, T, \{m_0\}]$ is called a (strict) coverability graph of N iff the following conditions hold:*

1. *States in S_C are vectors $\mu : P \longrightarrow \mathbf{N} \cup \{\omega\}$;*
2. *$m_0 \in S_C$;*
3. *If $\mu \in S_C$, and t is a transition s.t. $W([p, t]) \leq \mu(p)$ for all $p \in P$, then there is a state $\mu' \in S_C$ s.t. $[\mu, t, \mu'] \in E_C$;*
4. *If $[\mu, t, \mu'] \in E_C$ then $\Omega\mu \subseteq \Omega\mu'$ and there is a finite transition sequence w s.t. w can be executed at $\mu + \Delta t$, $\Delta w[p] = 0$ for $p \in P \backslash \Omega\mu'$, $\Delta w[p] > 0$ for $p \in \Omega\mu' \setminus \Omega\mu$, and $\mu'(p) = \mu(p) + \Delta t(p)$ for $p \in P \setminus \Omega\mu'$;*
5. *$[S_C, E_C, T]$ is connected from m_0.*

If $\Omega\mu = \Omega\mu'$ then the empty sequence can be used as w in the last item of the definition. μ' is then just the t-successor of μ as in the normal state space. A nonempty w is used to justify the introduction of new ω ($\Omega\mu \subset \Omega\mu'$). Since w can be executed at $\mu + \Delta t$, and leads to a greater marking, w can

be executed infinitely often from $\mu' + \Delta t$ producing an increasing sequence of markings. μ' corresponds then to $\mu' + \Delta t$ after an introduction of new ω in all components where w increases strictly. In the construction by Karp and Miller, w is determined as the sequence that leads from an ancestor of $\mu + \Delta t$ to $\mu + \Delta t$ in the search tree that is smaller than $\mu + \Delta t$. Fig. 7.2 depicts the Karp/Miller graph of the system in Fig. 7.1. The sequences w justify new ω introductions, are shown in parenthesis.

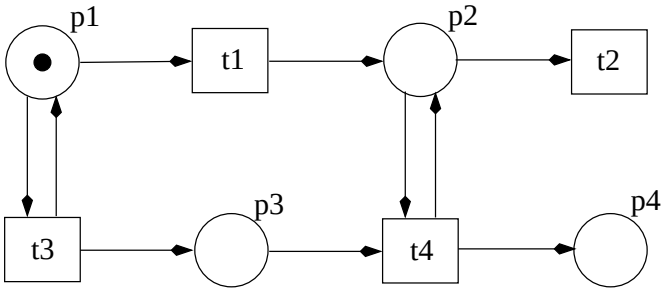

Figure 7.1: An infinite state Petri net

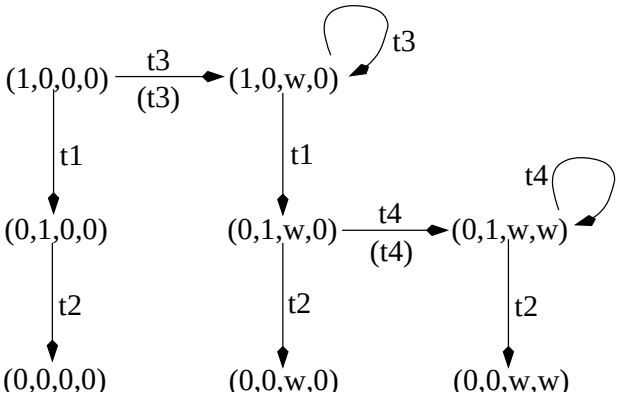

Figure 7.2: Karp/Miller coverability graph of the net in Fig. 7.1. For each edge where Ω-sets change, we annotated the required sequence w in parenthesis.

Definition 25 (Sloppy coverability graph) *Let $N = [P, T, F, W, m_0]$ be a Petri net. A transition system $[S_C, E_C, T]$ is a (sloppy) coverability graph of N iff the following conditions hold:*

1. *States in S_C are vectors $\mu : P \longrightarrow \mathbf{N} \cup \{\omega\}$;*

2. *There is a state* $\mu_0 \in S_C$ *s.t.* $\mu_0 \geq m_0$*;*

3. *For every* $\mu \in S_C$ *there is a sequence* $\{m_i\}_{i \in \mathbf{N}}$ *of reachable markings that is strictly increasing on all components in* $\Omega\mu$ *and has constant value* $\mu[p]$ *on every component* $p \notin \Omega\mu$*;*

4. *For every* $\mu \in S_C$ *and every transition* t *where* $W([p,t]) \leq \mu(p)$ *for all* $p \in P$*, there is a* $\mu' \in S_C$ *holding* $\mu' \geq \mu + \Delta t$*.*

The coverability graph defined by Finkel is minimal in the sense that it does not contain any two different states μ, μ' such that $\mu \leq \mu'$. This minimal coverability graph of the system in Fig. 7.1 is depicted in Fig. 7.3. This graph is not a strict coverability graph.

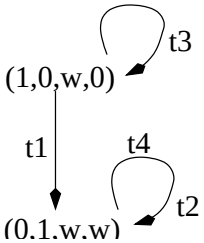

Figure 7.3: Finkel coverability graph of the net in Fig. 7.1

Every strict coverability graph is a sloppy coverability graph as well. We delay the proof of this fact because it is a corollary of one of our property preservation results.

Results on preservation of universal (ACTL*) properties rely on simulation relations between coverability graphs and the original state space. The simulation relation for strict coverability graphs is thereby tighter than the one for sloppy graphs, showing that more universal properties are preserved by Karp/Miller graphs than by Finkel graphs (at the prize that Karp/Miller graphs can be significantly larger than Finkel graphs).

Definition 26 (Simulation relations for coverability graphs) *Let* M *be the set of all markings* ($M = \{m \mid m : P \longrightarrow \mathbf{N}\}$)*, and* M^Ω *the set of all* ω*-markings* ($M^\Omega = \{\mu \mid \mu : P \longrightarrow \mathbf{N} \cup \{\omega\}\}$)*. Define* ρ_{st} *and* ρ_{sl} *(the strict and sloppy simulation relations) by*

$$m \rho_{st} \mu \text{ if and only if for all } p \notin \Omega\mu, m[p] = \mu[p];$$

$$m \rho_{sl} \mu \text{ if and only if } m \leq \mu.$$

Lemma 4 (ρ_{st} and ρ_{sl} are simulations) *Given a set of propositions preserved through ρ_{st} (ρ_{sl}, resp.). The state space of a Petri net ρ_{st}-simulates every strong coverability graph, and ρ_{sl}-simulates every sloppy coverability graph.*

Proof. Let $[S, E, T]$ be the state space, and $[S_C, E_C, T]$ a strict (sloppy, resp.) coverability graph of a Petri net N. By Item 2 of Def. 24 and Item 2 of Def. 25, in both coverability graphs there are states related to the initial state of the original state space.

Given an edge $[m, t, m']$ in the original state space and an abstract state μ related to m, we have to show that there is a state μ' in the coverability graph related to m', and an edge $[\mu, t, \mu']$. In the strict case, Item 3 of the definition assures the existence of a μ' such that $[\mu, t, \mu']$ is an edge. Since, by definition of ρ_{st}, we have $m(p) = \mu(p)$ for all $p \notin \Omega\mu$. Thus, $(\mu + \Delta t)[p] \geq (m + \Delta t)[p] = m'(p)$ for $p \notin \Omega\mu$. By Item 4 of the definition, μ' differs from $\mu + \Delta t$ at most by additional ω introduced in μ'. Thus, $m' \rho_{st} \mu'$.

In the sloppy case, Item 4 of the definition assures the existence of an edge $[\mu, t, \mu']$. Since $m \leq \mu$, we have $m' = m + \Delta t \leq \mu + \Delta t \leq \mu'$. Thus, $m' \rho_{sl} \mu'$. ♢

In order to extend the simulation results to a result on preserving a class of ACTL* formulas, we have to study the capabilities of the two relations to preserve atomic propositions. For the strict graph, ρ relates states only to abstract states by replacing components with ω, while it does not replace a finite value by another finite value. Thus, if an atomic proposition concerns only components that are not equal to ω in the abstract state, it is true of the abstract state if and only if it is true of every related concrete state. For other components, no definite preservation result is possible.

For the sloppy graph, concrete states may be related to an abstract state that has a finite, but larger value on some components. Thus, only atomic propositions of the forms "$p \leq k$"or "$p < k$"(p is a place, k a natural number) are true of the abstract state if and only if they are true of every related concrete state.

Since in both cases different states can preserve different atomic propositions, an ACTL* preservation result can only be established if we relax the validity of propositions in the abstract transition system: a proposition is true of an abstract state if and only if it is true of every related concrete state. Consequently, there may be atomic properties ϕ and coverability graph states μ where neither $\mu \models \phi$ nor $\mu \models \neg\phi$ which is not a serious problem

for the model checking techniques. At least, we can establish the following preservation results:

Corollary 5 (Coverability graphs preserve ACTL*) *An ACTL* formula that is true of a strict coverability graph and contains only atomic propositions of the forms "$p \leq k$", $p \geq k$", "$p < k$", "$p > k$", "$p = k$", and "$p \neq k$"where k is a natural number and p is a* bounded *place, is true of the concrete transition system as well.*

An ACTL formula that is true of a sloppy coverability graph and contains only boolean combinations without negation of atomic propositions of the forms "$p \leq k$"and "$p < k$"where k is a natural number and p is a place, is true of the concrete transition system as well.*

Among the properties that could be verified traditionally using a coverability graph were a few existential properties. For instance, if there is a state $(\omega, 0, 1, \omega)$ in the coverability graph, and k is a natural number, then there is a state $(k_1, 0, 1, k_2)$ reachable in the original state space where $k_1 > k$ and $k_2 > k$. In other words, the CTL formula $\mathbf{EF}p_1 > k \wedge p_4 > k$ can be verified on the coverability graph for arbitrary k. In the sequel, we develop an approach to the verification of existential properties on the coverability graph that includes more existential CTL properties. The rules for deriving properties have to be incomplete, though, since not all existential CTL properties are preserved by the coverability graph construction.

The difficulty in verifying existential properties on coverability graphs is that we have to prove that the existence of some path in the coverability graph implies the existence of a path in the original state space (for universal properties, we had to establish the reverse connection). Among the (original) states represented by a node in a coverability graph are, however, states having only few, maybe zero tokens on places where the node has value ω, so transitions may be disabled in those original states but enabled in the coverability graph state. To overcome this problem, we introduce the concept of limit-satisfiability that is based on the fact that for every state in a coverability graph there is a sequence of reachable markings converging to it.

Definition 27 (Convergence of marking sequences) *A sequence $\{m_i\}_{i \in \mathbf{N}}$ of markings $m_i : P \longrightarrow \mathbf{N}$ converges to μ, $\mu : P \longrightarrow (\mathbf{N} \cup \{\omega\})$, iff $m_i \rho_{st} \mu$ for all $i \in \mathbf{N}$ and for each $k \in \mathbf{N}$ there is a $n \in \mathbf{N}$ such that for all $j > n$ $\mu(p) = \omega$ implies $m_j(p) > k$.*

A converging sequence has constant values on places p where $\mu(p)$ is finite, and is monotonously increasing (not necessarily strictly increasing) on all places p where $\mu(p) = \omega$.

Proposition 6 ([KM69, Fin90]) *If μ is a node occurring in a strong or a sloppy coverability graph then there exists a sequence of reachable markings converging to μ.*

Based on these results, we define a new satisfiability relation for CTL formulas and coverability graphs:

Definition 28 (Limes-satisfiability) *A state μ, $\mu : P \longrightarrow (\mathbf{N} \cup \{\omega\})$, limit-satisfies a CTL formula ϕ ($\mu \models_{\lim} \phi$), iff for every sequence $\{m_i\}_{i \in \mathbf{N}}$ of markings converging to μ there is a k such that for all $j > k$, $m_j \models \phi$.*

We require that for each converging sequence, almost all members satisfy the formula. For example, definitions of convergence and limit-satisfiability imply that $(\omega, 1, 0, \omega) \models_{\lim} (p_1 > 1782 \wedge p_2 < 2 \wedge p_4 \neq 132687)$. Unlike the simulation based approach, limit-satisfiability permits some statements about places marked ω.

For a systematic approach to limit-satisfiability, we study it first for simple comparisons, then for boolean combinations of formulae, and finally for formulae containing temporal operators. For propositions of the form $p = k$, $p \neq k$, $p \leq k$, $p < k$, $p \geq k$, and $p > k$, the results summarized in Tab. 7.1 can be proven immediately from the two definitions above.

Table 7.1: Limes-satisfiability of atomic propositions

$\mu(p)$	$\in \mathbf{N}, = k$	$\in \mathbf{N}, < k$	$\in \mathbf{N}, > k$	$\mu(p) = \omega$
$\mu \models_{\lim} p = k$	yes	no	no	no
$\mu \models_{\lim} p \neq k$	no	yes	yes	yes
$\mu \models_{\lim} p \leq k$	yes	yes	no	no
$\mu \models_{\lim} p < k$	no	yes	no	no
$\mu \models_{\lim} p \geq k$	yes	no	yes	yes
$\mu \models_{\lim} p > k$	no	no	yes	yes

There may be other properties where limit-satisfiability can be established, especially if they concern only places where μ is not equal to ω. There are as well properties where in general we cannot conclude anything about

limit-satisfiability. For instance, the proposition $p < q$ about two places p and q labeled ω may or may not be true of a converging sequence.

Limes-satisfiability is closed under conjunction and disjunction.

Theorem 14 *Let ϕ and ψ be CTL formulas and $\mu : P \longrightarrow (\mathbf{N} \cup \{\omega\})$. Then $\mu \models_{\lim} \phi \wedge \psi$ if and only if $\mu \models_{\lim} \phi$ and $\mu \models_{\lim} \psi$. Furthermore, $\mu \models_{\lim} \phi$ or $\mu \models_{\lim} \psi$ implies $\mu \models_{\lim} \phi \vee \psi$.*

Proof.

Let $\{m_i\}_{i \in \mathbf{N}}$ be a sequence converging to μ. If $\mu \models_{\lim} \phi$ and $\mu \models_{\lim} \psi$ then there are numbers k_1 and k_2 such that for all $i > k_1$, $m_i \models \phi$ while for all $j > k_2$, $m_j \models \psi$. Thus, for all $l > max\{k_1, k_2\}$, $m_l \models \phi \wedge \psi$. If $\mu \models_{\lim} \phi \wedge \psi$, there is a number k such that for all $j > k$, $m_j \models \phi \wedge \psi$. Hence, for all $j > k$, $m_j \models \phi$ and $m_j \models \psi$.

The implication for disjunction can be proven similarly. ♢

Negation can be dealt with by using de Morgan's rules for removing them in front of other boolean operators. Negation in front of atomic proposition can be removed since our considered set of propositions is closed under negation.

We are now going to derive rules for deducing temporal properties from a coverability graph. The rules propagate validity of a temporal property from successor states to predecessor states and can thus be integrated into the backtracking phase of depth first coverability graph generation. As the first operator, we study **EF**.

Theorem 15 (Propagation of E F ,[Sch99a]) *Let μ and μ' be nodes in a strict coverability graph connected by an edge $[\mu, t, \mu']$. Let ϕ be a CTL formula. If $\mu' \models_{\lim} \mathbf{EF}\phi$ then $\mu \models_{\lim} \mathbf{EF}\phi$.*

The idea of the proof is to establish a path from almost all states of a sequence converging to μ to states that form a sequence converging to μ'. Since almost all elements of the second sequence satisfy $\mathbf{EF}\phi$, so do the connected states from the first sequence.

Proof. Let $\{m_i\}_{i \in \mathbf{N}}$ be a sequence of reachable states converging to μ. By definition of strict coverability graphs, t is enabled for almost all members of $\{m_i\}_{i \in \mathbf{N}}$. Consider first the case where $\Omega\mu = \Omega\mu'$. Then $\{m_i + \Delta t\}_{i \in \mathbf{N}}$ is easily shown to be a sequence converging to μ'. By assumption, almost

all members of that sequence satisfy $\mathbf{EF}\phi$. Hence, almost all members of $\{m_i\}_{i\in\mathbf{N}}$ can reach a state satisfying $\mathbf{EF}\phi$ and therefore satisfy $\mathbf{EF}\phi$ themselves.

Let now $\Omega\mu \subset \Omega\mu'$. By definition of strict coverability graphs, there is a sequence w of transitions s.t. w is executable at $\mu + \Delta t$, $\Delta w(p) = 0$ for $p \notin \Omega\mu'$, and $\Delta w(p) > 0$ on $p \in \Omega\mu' \setminus \Omega\mu$. We distinguish two subcases concerning the behavior of w on $\Omega\mu$. Assume first $\Delta w(p) \geq 0$ for all $p \in \Omega\mu$. In this case, if w is executable at some state, so is w^i (the concatenation of i copies of w) for any i. Thus, for almost all i, sequence tw^i is executable at member m_i of the sequence $\{m_i\}_{i\in\mathbf{N}}$. The resulting sequence $\{m_i + \Delta t w^i\}_{i\in\mathbf{N}}$,$tw$ is executable at m_i is converging to μ' since the number of tokens on places in $\Omega\mu$ is increasing (since it is increasing in $\{m_i\}_{i\in\mathbf{N}}$ and not decreasing by w^i), is increasing on $p \in \Omega\mu' \setminus \Omega\mu$ (since it is constant in the m_i and strictly increasing through the increasing number of occurrences of w), and equal to $m_i + \Delta t$ on $p \notin \Omega\mu'$ (since $\Delta w(p) = 0$ for those places).

Assume now that there is a place $p \in \Omega\mu$ where $\Delta w(p) < 0$. Then, w can be executed only finitely often at any marking m (at most $\frac{m(p)}{\Delta w(p)}$ times). Let n_i be the maximum number of times w can be executed at $m_i + \Delta t$ (let $n_i = 0$ if t is disabled at m_i). The sequence $\{n_i\}_{i\in\mathbf{N}}$ is diverging since t is enabled at almost all m_i, the number of tokens on places in $\Omega\mu$ is diverging, and other places cannot restrict the number of occurrences of w. Hence, sequence $tw^{\lfloor \frac{n_i}{2} \rfloor}$ is executable at almost all members m_i of sequence $\{m_i\}_{i\in\mathbf{N}}$. This defines an infinite sequence $\{m_i + \Delta t w^{\lfloor \frac{n_i}{2} \rfloor}\}_i \in \mathbf{N}$, tw^{n_i} is executable at m_i. This sequence is converging to μ': it is increasing on all places $p \in \Omega\mu$ where $\Delta w(p) \geq 0$ since $\{m_i\}_{i\in\mathbf{N}}$ is increasing and w does not decrease the number of tokens; it is increasing on $p \in \Omega\mu$ where $\Delta w(p) < 0$ since w is executable at least another $\lfloor \frac{n_i}{2} \rfloor$ times after execution of $tw^{\lfloor \frac{n_i}{2} \rfloor}$; it is increasing on $p \in \Omega\mu' \setminus \Omega\mu$ since $\{n_i\}_{i\in\mathbf{N}}$ is diverging and $\Delta w(p) > 0$ there; and it is equal to $m_i + \Delta t$ on $p \notin \Omega\mu'$ since $\Delta w(p) = 0$ on those places. Since the sequence just constructed converges to μ', it limit-satisfies $\mathbf{EF}\phi$ (by assumption). Consequently, almost all m_i can reach a state satisfying $\mathbf{EF}\phi$ and therefore satisfy $\mathbf{EF}\phi$ themselves. $\diamondsuit$

Using this theorem, we can propagate validity of $\mathbf{EF}\phi$ through a strict coverability graph. For a base case, observe that $m \models \phi$ implies $m \models \mathbf{EF}\phi$ for arbitrary states, so $\mu \models_{\lim} \phi$ implies $\mu \models_{\lim} \mathbf{EF}\phi$ for arbitrary nodes of a coverability graph.

Example. Consider a predicate of the form $\mathbf{EF}p > k$, for a place p and a natural number k. This predicate is true if and only if there is a state μ in the strict coverability graph where $\mu(p) > k$. We have therefore $\mu \models_{\lim} p > k$ since either $\mu(p)$ is a natural number and every member m of a sequence converging to μ has $m(p) = \mu(p)$, or $\mu(p) = \omega$, then at least almost all members of a sequence have values greater than k. From $\mu \models_{\lim} p > k$ we have $\mu \models_{\lim} \mathbf{EF}p > k$. Since the coverability graph is connected from m_0, we can use the above theorem to get $m_0 \models_{\lim} \mathbf{EF}p > k$. Since m_0 does not contain ω's, $m_0 \models_{\lim} \mathbf{EF}p > k$ implies $m_0 \models \mathbf{EF}p > k$.

If there is no state μ in the coverability graph with $\mu(p) > k$ then all members m of all sequences converging to states in the coverability graph have $m(p) \leq k$. Thus, every state in the coverability graph satisfies $\neg p > k$, and the ACTL formula $\mathbf{AG}\neg p > k$ is true of the coverability graph. By the result on ACTL* preservation, the original state space satisfies $\mathbf{AG}p > k$ as well, and does therefore not satisfy $\mathbf{EF}p > k$. Thus, coverability queries can always be answered using our set of rules. Similarly, most properties previously known to be preserved by coverability graphs turn out to be consequences of the rules provided so far.

Consider now the case $\mathbf{E}(\phi \mathbf{U} \psi)$. Trying to apply the same arguments as for **EF** shows one difficulty: For linking elements in a sequence converging to μ with elements in a sequence converging to μ', we used transition sequences where w, the sequence pumping tokens on the fresh Ω-places, is executed arbitrarily often. For establishing a result for an until-formula, ϕ must hold throughout these sequences. Fortunately, we can assume that w is explicitly available. In the Karp-Miller construction, w is a sequence starting at some ancestor of μ sharing the same Ω-set as μ, and consists of exactly the transitions between that ancestor and μ in the search tree. Thus, a single pointer from μ to that ancestor suffices to get access to the whole w. Now, the easiest way to assure that ϕ holds throughout arbitrarily long sequences made of w is that transitions in w are invisible to w (occurrence of a transition in w does not alter any atomic proposition occurring in w. Under these assumptions, it is easy to repeat the proof for **EF** for showing:

Theorem 16 (Propagation of $\mathbf{E}(\phi \mathbf{U} \psi)$) *Let μ and μ' be nodes in a strict coverability graph connected by an edge $[\mu, t, \mu']$. Let w be the sequence required in the definition for strict coverability graphs for letting $[\mu, t, \mu']$ be an edge. Let ϕ and ψ be CTL formulas. Assume, no transition in w can change*

values of atomic propositions in ϕ in any reachable state. Let $\mu' = \mu + \Delta t$ or $\mu + \Delta t \models_{\lim} \phi$. If $\mu' \models_{\lim} \mathbf{E}(\phi \mathbf{U} \psi)$ then $\mu \models_{\lim} \mathbf{E}(\phi \mathbf{U} \psi)$.

Since $\mu \models_{\lim} \psi$ implies $\mu \models_{\lim} \mathbf{E}(\phi \mathbf{U} \psi)$, we have again sufficient material to start propagation.

Dealing with $\mathbf{EX}\phi$ is simple. Instead of looking at the successor in the coverability graph, we can just look at the sequence $\mu + \Delta t$.

For $\mathbf{EG}\phi$, the situation is more complicated. Though we can again use the trick with invisible transitions for propagating valid **EG**–formulas through a strict coverability graph, ...

Theorem 17 (Propagation of E G) *Let μ and μ' be nodes in a strict coverability graph connected by an edge $[\mu, t, \mu']$. Let w be the sequence required in the definition for strict coverability graphs for letting $[\mu, t, \mu']$ be an edge. Let ϕ be a CTL formula. Assume, no transition in w can change values of atomic propositions in ϕ in any reachable state. Let $\mu' = \mu + \Delta t$ or $\mu + \Delta t \models_{\lim} \phi$. If $\mu' \models_{\lim} \mathbf{EG}\phi)$ then $\mu \models_{\lim} \mathbf{EG}\phi$.*

... it is more difficult to find a first state that satisfies $\mathbf{EG}\phi$. A witness for an **EG** formula must form an infinite sequence of states satisfying ϕ. In a transition system, we detect witnesses usually as cycles. Consider, for example, the self loop with $t4$ in state $(0, 1, \omega, \omega)$ in Fig. 7.2. This is a cycle where all states hold $p2 = 1$. Nevertheless, $\mathbf{EG}p2 = 1$ is false of every member of any sequence converging to $(0, 1, \omega, \omega)$. The reason is that at the moment $p2$ is marked, no new tokens can arrive on $p3$. $t4$ as the only enabled transition maintaining $p2 = 1$, however, consumes tokens from $p3$, and can thus not be executed infinitely often. Thus, eventually $t2$ becomes the only enabled transition and $p2 = 1$ does not hold any longer. We must therefore distinguish between decrescent and non-decrescent cycles in the coverability graph. A cycle in the coverability graph is non-decrescent iff the sequence of transitions u establishing the roundtrip in this cycle has $\Delta u(p) \geq 0$ for all places p. Note that in all states forming a cycle, the Ω-sets must be identical, so pumping sequences can be ignored in considerations on cycles. If a cycle is non-decrescent, it is easy to see that infinite paths through members of the cycle can be established from every sufficiently large member of sequences converging to one of the members of the cycle. Thus,

Theorem 18 (seed for EG-propagation) *If all members of a non-decrescent cycle in a strict coverability graph limit-satisfy ϕ, then all these members limit-satisfy $\mathbf{EG}\phi$.*

7.2 Algorithms

According to [KM69], strict coverability graphs can be implemented as follows. Whenever a new marking μ is encountered, every marking μ' on the path back to the initial state is is checked for $\mu' < \mu$. If such a state is found, μ is set to ω in all positions p where $\mu'(p) < \mu(p)$. This whole process can be understood as part of the computation of μ, thus strict coverability graph computation can be integrated in any search algorithm where the search tree is explicitly available.

In Sec. 3.3.2, we argued that we can get more flexibility in implementing a data structure for the actual state space by restricting the set of operations to be performed on this data structure. We pointed out that, given a state s, it is desirable to have as few as possible information to be retrieved from s, so that other information can be compressed, or even thrown away. In particular, we would like to be able to replace the full image of a state (the complete marking vector) by a less space consuming "fingerprint", that is smaller yet identifying a state uniquely (see next chapter for such a fingerprint mechanism). A brute force implementation of coverability graphs as sketched above requires full knowledge of the complete vector μ', for all μ' on the path back to the initial state (in worst case).

Fortunately, we *are* able to implement coverability graph generation in connection with fingerprint techniques. As the only additional information to be stored in each state s , we need to know:

- Which places have become ω when s was created;
- Which values on those places were replaced by ω;

For most systems, introduction of new ω is a rare event, so the information we require to store does not generate significant space overhead.

With the stored information, we can start with a copy of μ, and successively generate predecessor states of μ by firing transitions backwards. The transition to be fired, as well as a handle (pointer) to the required information, can be found on the stack. If that state carries information about introduction of ω, the stored information can be used to restore the value of the state. This way, we can exactly restore the vector corresponding to all predecessor states, without retrieving this vector from the explicit data structure representing the state space. The involved operations are cheap (as firing forward or backward involves only few places of a distributed system).

Implementation of sloppy coverability graphs is more involved. [Fin90] contains an implementation of a minimal coverability graph which is the implementation of the sloppy coverability graph with least number of states. The main differences to the construction in [KM69] are

- If a predecessor μ' is less than the current marking μ, then ωs are introduced in μ' rather than in μ. All successors of μ' in the search tree are thrown away.
- If there is a marking μ' that is smaller than the current marking μ and not an ancestor in the search tree, then μ' and all its successors in the search tree are thrown away.
- If there is a node μ' anywhere in the search tree that is larger than the current node μ then μ is thrown away.

This algorithm differs significantly from other state space generation techniques. States need to be removed from the state space during state space generation, and we need to check the state space for markings that are less or greater than the current marking, not just the marking itself. The technique requires therefore specific data structures. In LoLA, we decided not to implement minimal coverability graphs. We have therefore no experimental results on this technique. For an implementation of minimal coverability graphs in the Petri net tool INA [RS98], we refer to the diploma thesis [L95]. This thesis contains an example where a sloppy coverability graph is much smaller than a strict coverability graph (some hundred states vs. hundreds of thousands). On finite state systems, both techniques differ only marginally. While the Karp-Miller construction yields always the complete state space of a finite state system, there are some rare situations where Finkel's construction can save a state or two.

7.3 Performance

For infinite state systems, the coverability graph technique is invaluable, since it reduces an infinite state system to a finite state system. For finite state systems, the size of a coverability graph is, with few exceptions in the Finkel construction, the same as the full state space, but takes more time since we check, in every state, for covered ancestors.

Table 7.2: Behaviour of Karp/Miller coverability graph generation on finite state Petri nets; depth first versus breadth first strategy

	time (depth first)	time (depth first)
PH 7	2.283	0.099
PH 8	29.968	0.301
PH 9	422.819	1.043
PH 10		3.764
PH 12		54.908
DA 10	12.157	0.615
DA 12	336.534	3.831
DA 14		24.515

The tables show that the run time penalty for coverability graphs is more significant in depth first search. The reason is that depth creates much longer search stacks than breadth first search, so search for ancestors concerns a larger number of states. It is therefore recommandable to apply coverability analysis with breadth first search than with depth first search which is somewhat surprising.

7.4 Compatibility

Coverability graphs can be used in combination with simple stubborn set methods. Finiteness and correctness of the coverability graph construction are proven solely based on the transition system level, so the original transition system can be replaced by a reduced transition system. There are UP set stubborn set approaches to boundedness and to the existence of dead transitions (transitions that are never enabled) [Sch99b]. Both properties are preserved by coverability graphs, so they can be verified by a stubborn reduced coverability graph construction. For advanced stubborn set methods, we have the problem that coverability graphs make sense only for infinite state Petri nets, where constructions such as the removal of ignored transitions have not yet been studied in sufficient detail.

For symmetry reduction, we can again establish correctness of the combined approach by viewing the jointly reduced transition system as a coverability graph construction on a symmetrically reduced transition system. However, technical details are more involved in the symmetry case: If there is

a path leading from marking μ' to marking μ in the unreduced transition system, and $\mu' < \mu$, then the corresponding states in the symmetrically reduced system are not necessarily comparable since one of them (or both) could have been replaced by a symmetric image. This problem occurs when canonicalization is used as the symmetry integration technique. For integration of symmetries by iterating states or symmetries, markings in the search *tree* are connected by usual reachability, so the coverability test can be executed as in the original transition system.

Concerning exploration strategies, our examples showed that using breadth first search is strictly more efficient than using depth first search in the construction of coverability graphs. With distributed search, the fact that we need to frequently examine states on the path from the initial marking to the current marking creates a prohibitively large communication overhead.

7.5 Discussion

Coverability graph construction was an early technique for the verification of infinite state systems. It can be seen as an example of a symbolic verification technique, but its construction, at least in the strict case, is so close to other explicit verification techniques that we decided to classify it as an explicit method. The possibility to combine coverability graph construction with stubborn sets and/or symmetries proves the close relationship. Coverability graph construction is a rather time consuming task, and there are still limited results on the preservation of properties. However, the capability of representing infinite state systems finitely makes coverability graphs a strong verification tool. The technique is closely tied to Petri nets, since it relies in large parts on the linearity and monotonicity of the Petri net firing rule (if a sequence can be executed in some state, it can be executed at every larger sequence, too, and leads to a larger final state). This monotonicity is a result of a strictly resource oriented view on systems that is not present in most other system description formalisms.

A major weakness of our propagation rules is that existential properties rely on a different abstract notion of satisfaction (limit-satisfaction). Since universal properties can cope only with usual satisfaction (all concrete states related to an abstract state need to satisfy the predicate), it is not possible to verify a property where an existential property is subformula of a universal property. Home properties, for instance, can therefore not be verified with

our tools. The other way round, we can cope with a universal formula being a subformula of an existential formula.

Karp/Miller and Finkel coverability graphs are essentially different. While the Karp/Miller graph can be substantially larger than a corresponding Finkel graph, it has a lot more capabilities to infer properties of the original system. While, with a Finkel graph, only a class of universal properties can be deduced, the class of universal properties verifiable with Karp/Miller graphs is significantly larger, and several existential properties can be derived, too.

Chapter 8

Linear algebraic reduction

Besides coverability graphs, the linear algebraic invariant calculus [Pet73, LS74, GL83, Jen81, RV87, Sch96b] is another class of techniques closely related to Petri nets. It relies on the nature of Petri nets as vector addition systems.

There are two kinds of linear invariants. The first one, place invariants, assign a numerical value to each state that remains constant under transition occurrence (i.e. it assigns the same value to reachable states as for the initial state). The second one, transition invariants, characterize transition sequences that, if enabled, lead from any marking to itself (form a cycle).

Place invariants have received a lot more attention than transition invariants which is due to the fact that they provide a simple and efficient overapproximation of the set or reachable states (as the set of states to which the invariant assigns the same value as to the initial state). Since invariants do not provide any information about connectivity between states, it is mainly reachability properties $\mathbf{AG}\phi$ that can be verified using a place invariant by verifying that a state violating ϕ must receive a different value than the initial state of the system. This is one of the most efficient and most successful structural verification techniques.

For transition invariants, only much weaker links to the system behavior are known. Most prominently, there is a theorem stating that a live and bounded Petri nets has a strictly positive transition invariant.

In the sequel, we are going to employ invariants for explicit state space verification. Place invariants can be used for compressing states. Transition invariants can be used to reduce the number of states to be stored without reducing the number of visited states.

8.1 Theory

Given a Petri net $N = [P, T, F, W, m_0]$ and a transition $t \in T$, define vectors t^-, t^+, and Δt (all having P as their index set) as follows: $t^-(p) = W([p, t])$, $t^+(p) = W([t, p])$, and $\Delta t = t^+ - t^-$. From the definition of reachability, we have that $m \xrightarrow{t} m'$ if and only if $m \geq t^-$ and $m' = m + \Delta t$.

Let C be a matrix with T indexing the columns, and P indexing the rows such that the column $C(., t)$ corresponds to Δt. This matrix C is called the *incidence matrix* of the Petri net N. Let further, for a transition sequence w, $\Psi(w)$ (the Parikh vector of w) be the vector assigning to each $t \in T$ the number of t's occurrences in w. Then the above relation, extended to transition sequences leads to

$$\text{if } m \xrightarrow{w} m' \text{ then } m' = m + C \cdot \Psi(w).$$

The equation at the right is known as the Petri net *state equation*.

A P–indexed row i of integers such that $i \cdot C = \underline{0}$ (where $\underline{0}$ is a T–indexed column of zeros) is called a *place invariant*. Multiplying i from the left to the state equation and eliminating the zero term $i \cdot C \cdot \Psi(w)$ leads to $i \cdot m = i \cdot m'$. This equality says that all reachable states of a Petri net have the same inner product with a place invariant. For state space based verification, we rely particularly on the capability of place invariants to express the value of some component of a state in terms of the other components. Let i be a place invariant and p a place such that $i(p) \neq 0$, m_0 be the initial state, and m any reachable state. Then, $i \cdot m = i \cdot m_0$ holds by the above considerations and can be rewritten to

$$m(p) = \frac{i \cdot m_0 - \sum_{p' \in P \setminus \{p\}} i(p') \cdot m(p')}{i(p)}$$

This means that knowing i, $m(p)$ does not need to be stored in the search structure since it can be reconstructed from the remaining components. Furthermore, two reachable markings are equal if and only if they are equal in all components of $P \setminus \{p\}$.

In general, a Petri net has several place invariants. Thus, the sketched approach can be applied to several places simultaneously. Consider a partition of P into a set of significant places P_{sig} and a set of insignificant places P_{insig} ($P_{sig} \cap P_{insig} = \emptyset$, $P_{sig} \cup P_{insig} = P$) such that for every $p \in P_{insig}$ there is a place invariant i_p such that $i_p(p) \neq 0$ and $i_p(p') = 0$ for all $p' \in P_{insig} \setminus \{p\}$.

Given such a partition, all components in P_{insig} of reachable states can be reconstructed from values in P_{sig} and a place invariant. Furthermore, two reachable states are equal if and only if they are equal on P_{sig}. Observe that the maximum size of P_{insig} equals the number of linear independent place invariants of the Petri net.

Under certain circumstances, the only operations to be performed on a search structure containing the already computed states are search for a state and insertion of a state. We discuss these circumstances in Sec. 8.4 of this chapter. In particular, as pointed out in Sec. 4.3.1, states in the search structure are irrelevant for restoring states upon backtracking from search branches, since those restorings can be implemented by firing transitions backwards. Since for comparing two states the components of significant places suffice, it is sufficient to only include the projection of a state to P_{sig} into the search structure. Interestingly, it is not necessary to know the actual invariants i_p for this approach as long as a feasible partition into significant and insignificant places is known. This partition can be represented without significant memory. In contrast, the size of states to be stored shrinks by as many components as there are linear independent place invariants. Additionally, comparison and insert operations in the search structure are performed on smaller vectors which promises better run time performance. In the next section we devise an algorithm to compute the required partition of P which is the only notable problem left for this state compression approach.

A *transition invariant* is a T–indexed integer vector i such that $C \cdot i = \underline{0}$ (where $\underline{0}$ is a P–indexed row of zeros). We can deduce from the state equation that, given a transition sequence w s.t. $m \xrightarrow{w} m$ for some state m, w's Parikh vector $\Psi(w)$ is a transition invariant. The other way round, if w has a transition invariant as its count vector and can be executed at some state m, it leads to m again (forms a cycle). Cycles are of interest for state space verification since for termination of search algorithms it is sufficient to store (and not explore for a second time) just one state of each cycle while other states can be encountered more than once. Additionally, cycles are instrumental for advanced partial order reduction techniques.

Let T_{close} be a set of transitions such that every nonzero transition invariant has at least one nonzero component in T_{close}. By the above considerations, every cycle in the state space involves the occurrence of a transition in T_{close}. Expressed differently, every cycle in the state space involves a state where elements of T_{close} are enabled, and only such states need to be shipped to the

search structure when explored.

In system description languages where the system is composed of explicit transition systems, transition invariants can be replaced by an investigation of cycles in the components. This has been exercised for instance in [LLPY97] which actually inspired the transition invariant approach presented here.

Instead of invariants, the state equation itself has been used to serve as a structural verification techniques. It can be immediately used as a necessary condition for reachability—if the equation

$$m' = m + C \cdot x$$

does not have nonnegative integer solutions, m' cannot be reachable from m. The test is usually weakened to rational solutions, due to a better complexity of that problem, and solved as a linear optimization problem. Thus, x can be assumed to be minimal, and it tends to be integer.

We propose, for the case that the state equation based test returns with a minimal integer solution, a heuristically narrowed search for reachability of m'. m' can thereby be specified partially, in this case only equations concerning specified components of m' form the system of equations that needs to be solved.

Our heuristics is based on the following thoughts: If m' is reachable from m then the Parikh-vector of the sequence from m to m' is a solution of the state equation. Shorter sequences yield smaller solutions. Though it is not guaranteed that the smallest sequence from m to m' corresponds to the minimal non-negative solution of the state equation, it is still the case often enough to explore that option first, before running into deeper regions of the state space. That is, we propose to start depth first search, but exploring, from m, only those sequences whose Parikh-vector is less or equal to the considered minimal solution of the state equation. This imposes a natural depth restriction. Only if search in the restricted search space fails, we gradually weaken our depth restrictions. This way, we do not save states when m' is unreachable. However, since the state equation *does* have a solution, m' is likely to *be* reachable. And if m' is reachable, we may have explored promising regions of the state space earlier than by usual depth first search.

8.2 Algorithms

For place invariant based compression, we need to compute a partition of P into P_{sig} and P_{insig}. For transition invariant based reduction, we need to find a set T_{close} of transitions. Interestingly, for neither of these problems it is necessary to actually compute invariants. It is sufficient to transform the set of equations that defines the respective kind of invariants, into upper triangular form. This can be done by multiplying equations with nonzero integers, adding equations to others, and changing the order of equations. A linear system $A \cdot \underline{x} = \underline{0}$ with A being a $m \times n$ integer matrix, $\underline{x}$ an n-dimensional vector of variables, and $\underline{0}$ an m-dimensional vector of zeros) is in upper triangular form, if for every row index $i < m$ and every column index $j \leq n$, $A[i,1] = A[i,2] = \cdots = A[i,j] = 0$ implies $A[i+1,1] = A[i+1,2] = \cdots = A[i+1,j+1] =$, i.e. the number of of leading zeros in A is strictly monotonously increasing with increasing row index.

An upper triangular form defines a partition of the variables into *head* variables and *tail* variables. x_j is a head variables iff there is a row i in A such that $A[i,j] \neq 0$ and $A[i,k] = 0$ for all $k < j$. That is, a head variable corresponds to the leftmost nonzero entry of some equation in the upper triangular form. If x_j is not a head variable, it is a tail variable. If we skip rows in A that have only zero entries (are therefore tautologies), every equation has at least its own head variable as nonzero entry. Other nonzero entries correspond to larger rows' head variables, and tail variables. Any partial assignment of values to all variables but an equation's head variable can obviously be uniquely extended to a solution for that equation by assigning some (zero or nonzero) value to the head variable.

Thus, starting at the bottom row and proceeding upwards, any assignment to the tail variables can be completed uniquely to a solution of the full system of equations. In particular, for any assignment of 1 to some tail variable and 0 to all other tail variables, there is a solution to the system of equations. This solution is rational in the first place but can be transformed into an integer solution by multiplying with the greatest common denominator. The solutions obtained this way are linear independent (due to the 0/1 setting on the tail variables), and can generate every solution by linear combination (by rank considerations).

Example. Consider the following system of linear equations in upper

triangular form.

$$\begin{aligned} x_1 + 2x_2 - x_3 - x_4 + 5x_5 &= 0 \\ x_2 - x_4 &= 0 \\ x_4 + x_5 &= 0 \end{aligned}$$

x_1, x_2, and x_4, are head variables, x_3 and x_5 are tail variables. Assigning $x_3 = 1, x_5 = 0$ extends to a unique solution $x_4 = 0$ (bottom equation), $x_2 = 0$ (middle equation), and $x_1 = 1$ (top equation) of the whole system. Likewise, $x_3 = 0, x_5 = 1$ leads to a solution $x_4 = -1, x_2 = -1, x_1 = -4$.

Applying the above considerations to the place invariant context, variables correspond to places, A is the transposed of the incidence matrix C, and solutions to the system of equations are place invariants. Now, letting P_{sig} be the set of head variables of the upper triangular form of the transposed of C, and P_{insig} the tail variables, there is, for each tail variable, a solution that is nonzero on it, and zero on all other tail variables. Thus, this partition satisfies the requirements established in the previous section.

For transition invariants, A is the incidence matrix itself, variables correspond to transitions, and solutions are transition invariants. Assume that there is a transition invariant that is zero on all tail variables. Since this solution must be a linear combination of the solutions described above, and due to the 0/1 structure of those solutions on the tail variables, this transition invariant can only be the zero vector. Thus, every nonzero transition invariant has at least one nonzero entry in the tail variables which justifies setting T_{close} to the set of tail variables of the upper triangular form of the incidence matrix.

This concludes the discussion on the preprocessing steps for both invariant based compression and reduction techniques. Experiments show that the investigation of upper triangular forms is very time efficient. For the place invariant approach, we shall see that the preprocessing overhead is more than compensated by a faster processing of smaller states in the search structure. For the transition invariant approach, we have to pay with a tremendous time overhead for the saved states, due to multiple exploration of states that are not recorded in the search structure.

In order to alleviate the time overhead in the transition invariant approach, we chose to implement a controllable time/space tradeoff. By storing not only states that enable a transition in T_{close}, but also states that are found in a depth divisible by some configurable number k, we limit the

amount of multiple explorations of states. Experiments show that with reasonable space overhead, time consumption can be limited significantly. The transition invariant based depth first search algorithm is depicted in Fig. 8.1. It assumes that preprocessing has already assigned a feasible value to T_{close}.

Figure 8.1: Depth first search using transition invariant based reduction

```
1    var V: set of markings initial ∅;
2    var current: marking initial m_0;
3    var depth: integer initial 0;
4    procedure TStateGraph()
5    var t : transition;
6    var Enabled: set of transitions;
7    begin
8        Enabled := {t | t ∈ T ∧ current ≥ t^-};
9        if Enabled ∩T_close ≠ ∅ or depth ≡ 0 mod k then
10           V := V ∪{current};
11       fi
12       for t in Enabled do
13           current := current +t^+ − t^-; depth := depth + 1;
14           if current ∉ V then
15               TStateGraph();
16           fi
17           current := current +t^- − t^+; depth := depth - 1;
18       done
19   end.
```

Concerning the heuristical invocation of a minimal solution of the state equation into reachability analysis, we assume that preprocessing provides us with an integer transition vector $\underline{v}$. In the proposed implementation below, after not succeeding to reach a a state on a path corresponding to Parikh vector $\underline{v}$, we open the search window by increasing the values in $\underline{v}$ (paths that correspond to Parikh values smaller than $\underline{v}$ are explored anyway). The algorithm terminates if increasing the value of $\underline{v}$ does not yield new states which means that the full state space is explored. One can avoid exploring states multiple times in multiple iterations of the the algorithm by keeping

track of states with enabled transitions yet to be explored. As in the previous section m^* denotes the target marking.

8.3 Performance

Our PH n system has $5n$ places and $2n$ linear independent place invariants. Thus, reduction in terms of vector length of states to be stored is 40%. The system DA n has $6n + 1$ places and $3n + 1$ independent place invariants. Thus, the vector length is reduced to 50%. In several other examples, we got reductions by 30% through 50%. Concerning run time, we got the following results.

Table 8.1: Run time for place invariant compression

	PH10	PH11	PH12	DA14	DA16	DA18
states	59048	117146	531440	16398	65552	262162
time (sec) w/o red.	2.7	9.8	37.7	2.0	10.4	53.8
time (sec) w/ red.	2.1	7.7	30.3	1.3	6.7	35.8

The reported systems are tiny since we wanted to compare run times between place invariant reduction and brute force state space generation. In connection with other reduction techniques, the speed up is comparable. For the 10,000 philosophers system (50,000 places, 40,000 transitions) the invariant related preprocessing takes still less than 1 second. Storing the $20,000$ generators of the place invariants would have become a serious problem by itself.

For the transition invariant technique, it is interesting to study the impact of the heuristical parameter k that controls the amount of additionally stored states. A PH n system has n independent transition invariants. DA n has $2n$ independent transition invariants.

The first two tables show the limited capabilities of transition invariant based reduction without combining it with partial order reduction. The first table shows the impact of k. For the data base example, other values of k do not change the number of states which can be blamed to very tight cycles in that net.

The last table shows the behavior of transition invariant based reduction in connection with partial order reduction. We use deadlock preserving

Figure 8.2: Parikh vector guided state space exploration

```
1   var V: set of markings initial ∅;
2   var current: marking initial m_0;
3   var bound: T-vector initial v;
4   var newfound: boolean initial true;
5   procedure ParikhSearch()
6   var Enabled: set of transitions;
7   begin
8       V := V {current};
9       Enabled := {t | t ∈ T ∧ current ≥ t⁻};
10      for t in Enabled do
11          if bound[t] > 0 then
12              bound[t] := bound[t] - 1;
13              current := current +t⁺ − t⁻;
14              if current = m* then stop fi;
15              if current ∉ V then
16              newfound := true;
17                  ParikhSearch();
18              fi
19              current := current +t⁻ − t⁺; depth := depth - 1;
20              bound[t] := bound[t] + 1;
21          fi
22      done
23  end.
```

```
1   procedure ParikhReachability()
2   begin
3       while newfound do
4           ParikhSearch();
5           for all t ∈ T do
6               bound[t] := bound[t] + 1;
7           done
8       done
9   end.
```

Table 8.2: Run time for transition invariant based reduction—the PHi example

	PH 5	PH 6	PH 7	PH 8	PH 9
states w/o red.	242	728	2186	6560	19682
time (sec) w/o red.	0.03	0.04	0.08	0.23	0.76
states w/ red., $k = 5000$	160	530	1708	5417	16952
time (sec) w/ red., $k = 5000$	0.09	0.7	9.7	136.0	2177.6
states w/red., $k = 20$	186	591	1828	5664	17545
time (sec) w/ red., $k = 20$	0.05	0.1	0.36	3.19	10.8
states w/red., $k = 10$	201	629	1947	5984	18289
time (sec) w/ red., $k = 10$	0.04	0.07	0.19	0.59	1.8

Table 8.3: Run time for transition invariant based reduction—the DAi example

	DA 14	DA 16	DA 18
states	16398	65552	262162
time (sec) w/o red.	2.0	10.4	53.8
states w/ red. $k = 10$	16384	65536	262144
time (sec.) w/ red. $k = 10$	2.1	10.4	55.1

stubborn sets as the partial order reduction technique. The numbers show a significantly better performance of the transition invariance technique.

Table 8.4: Performance of the transition invariant method in connection with stubborn sets.

	PH 100	PH 200	DA 200	DA 300	DA 400
states (p.o. red. only)	29702	119402	401	601	801
time (sec) (p.o. red. only)	2.2	16.4	8.2	25.1	61.0
states both red., $k = 5000$	10311	41093	1	1	1
time (sec.) both red., $k = 5000$	45.3	395.3	13.0	26.8	64.6
states both red., $k = 20$	14502	59002	1	1	1
time (sec.) both red., $k = 20$	3.5	26.5	8.0	26.7	64.4
states both red., $k = 10$	17702	71402	1	1	1
time (sec.) both red., $k = 10$	2.8	21.4	8.3	26.7	64.0

8.4 Compatibility

The place invariant based compression technique replaces a state by a fingerprint from which the original state vector can not be restored without major

efforts (notice that we do not store the invariants for actually restoring the missing components). In the previous chapters, we discussed compatibility of the other reduction techniques with fingerprint like compression. We saw only few incompatibilities: one particular technique for implementing symmetrically reduced transition systems, and the generation of sloppy coverability graphs. The vast majority of reduction methods can be applied in combination with fingerprint techniques, and thus with the compression described above.

Since the transition invariant based technique only controls which markings are permanently stored, but not which successors are explored at a marking, there are no compatibility problems with simple stubborn set techniques. For advanced techniques, it requires future research to check whether the necessary information on strongly connected components can be retrieved from the reduced graph (the original edge relation is unavailable due to the removal of states). Simple partial order reduction techniques, however, increase the performance of the transition invariant technique by reducing the average branching factor in the state graph. Cycles in a graph reduced by partial order reduction, are definitely cycles in the original graph, so the transition invariant technique works correctly for partial order reduced graphs.

For symmetries, we need to be careful since cycles in a symmetrically reduced graph do not necessarily correspond to cycles in the original graph. The reason is that a marking can have an edge to a symmetric image of its actual successor marking. However, a cycle in the reduced graph can be extended to a cycle in the original graph since for every sequence executable in a state, a sequence consisting of equivalent transitions can be executed at a symmetric state. Thus, for every cycle in the reduced graph there is a cycle in the original graph (maybe longer) that consists of the transitions occurring in the reduced graph's cycle and transitions equivalent to them. Thus, enlarging U by all transitions that are equivalent w.r.t. symmetry to elements in U can guarantee that the enlarged U contains at least one transition for each cycle in the symmetrically reduced state graph.

Transition invariant based reduction and coverability graph reduction are incompatible since the not permanently stored states are not accessible for the covering test that is essential for coverability graph construction. Theoretically, the test can be reduced to the permanently stored elements, but the overhead would be unacceptable.

Heuristically narrowed search can only be applied to small number of

reachability problems. We have no running implementation, but first considerations [Sch00d] show that there is some synergy between heuristically narrowed search and simple stubborn set methods. Narrowed search limits the *number* of occurrences of transitions, stubborn sets limit the number of permutations of transitions to be explored with one and the same Parikh vector.

8.5 Discussion

The place invariant technique yields improvements in both space and time. It is compatible with many other reduction techniques. Thus, no special care needs to be taken when applying the method. The transition invariant based approach turns out to be infeasible as a standalone technique but can be valuable when applied in connection with partial order reduction. It requires that the parameter k that controls the space/time tradeoff of the technique be chosen carefully. Then, however, it can make the difference between fitting a state graph into the available memory or not. Our fingerprint technique retains more information of a state than usual hash compaction techniques. Those techniques [Hol91, Hol88, WL93, SD96] usually transform states into another domain such that they can no longer be identified uniquely (though they try to decrease probability of conflicts). In such settings, it is not always possible to get a definite answer to a verification problem. We think that it is worth looking for more powerful techniques with unique fingerprints.

Feasibility of both methods relies on the fact that we do not need the actual values of the invariants but only information that, in a generating set, certain elements are guaranteed to be 0 or nonzero. This way, no space is required for storing invariants, and the preprocessing can be reduced to just generating an upper triangular form out of the net's incidence matrix (or its transposed).

Besides that fact that the linear algebraic approach to invariance is already rather Petri net specific (even low level net specific), we benefit from invertible actions that enable us to backtrack to predecessor states without retrieving states from the state space data structure.

Part IV

Conclusions

Chapter 9

Comparison with other verification tools

9.1 SMV

SMV [McM93] is a family of CTL model checkers. It pioneered the use of symbolic model checking on the basis of binary decision diagrams [Bry86]. Several versions of SMV from different sources are available. Our results are based on Cadence SMV [McM02], reportedly among the fastest freely available versions. It is well known that the performance of SMV depends on a carefully chosen order in which state variables appear the the main data structure—the binary decision diagram. This order can be provided manually. Lacking the necessary experience, we cannot construct such an ordering. We do, however, use Cadence SMV's capabilities to compute an acceptable ordering heuristically (option `-f`), and to reorder variables while verification is running (option `-sift`). The latter option is reported to slow down verification. In our examples, the difference was insignificant.

As an input to SMV, we created a model in the SMV language from scratch, as opposed to generating input from a Petri net representation automatically. The latter is offered by the Petri net tool PEP [Gra97] but leads to models with a larger number of variables than usually occur in SMV models. The reason is that Petri nets have frequently several places (for instance one for presence and one for absence of a resource) where variable oriented languages have only one variable. In the philosophers example, two bits suffice to represent the local state of a philosopher while a usual Petri net model

would have four places for that purpose. The PEP translation would create one boolean variable per place, and thus feed SMV with an unnecessarily large model. In order to treat SMV fair, we decided to use a dedicated model that has, however, the same state space as the Petri net model used for the LoLA experiments.

Table 9.1: Performance of SMV on our running examples. In the philosophers example, we check mutually exclusive eating of two neighbours, and home of an eating state for a philosopher. For the DA system, we check exclusion between a reading and a writing operation as well as home of writing

	reachability	home
PH 10	1.045	1.057
PH 30	110.060	110.860
PH 50	1283.704	1274.398
DA 10	4.638	7.377
DA 30	773.716	946.553

The examples show that for the kind of examples dealt with in this thesis, explicit state space reduction can cope with state space explosion better than symbolic state space exploration. This is true because partial order reduction yields excellent results on distributed, asynchronously communication systems. We believe that it is not necessary to provide experimental evidence for the superiority of symbolic verification techniques in many other domains.

9.2 Spin

Spin is an explicit state model checker for LTL featuring a version of partial order reduction as well as several techniques for unique or non-unique state compression.

For the Spin input, we applied the same considerations concerning input as for SMV. Promela, the input language for Spin, has a concept of processes. Naturally, one would model a system such as the philosophers example as a set of processes. PEP creates, starting from a Petri net, only one monolithic process for the whole system. However, processes in Promela need to be forked from a single initial process. This way, additional states result where some already running processes interleave with the initialization process for

the remaining processes. It turned out that Spin yielded better results for the monolithic process created by PEP (and it is those results we report). Since we have only incomplete knowledge to which degree process structure influences Spin's partial order reduction approach, and given that we have only limited skills in writing Promela code, we need to interpret the experimental results carefully. It may be that the reported numbers are significantly below Spin's actual capabilities.

Table 9.2: Performance of Spin on a local reachability property

	PH 5	PH 10	PH 50
time	0.013	0.734	664.762

The results show that partial order reduction leads to significant savings, compared to brute force state space reduction. On the other hand, LTL preserving reduction methods cannot compete with dedicated stubborn sets for reachability properties.

For larger instances of the DA example, we did not manage to produce a reliable model in Promela (note, that the system requires to take a *set* of semaphores at a time).

9.3 Murϕ

Murϕ is an explicit state verification tool featuring scalarset symmetries. It does not use partial order reduction, and does not support other kinds of symmetry. Thus, among our running examples, only DA n is a candidate for verification with Murϕ.

Table 9.3: Performance of Murϕ on a system with dense symmetry.

	Murϕ (no symms)	Murϕ (symms)
DA 10	2.461	0.146
DA 12	45.400	0.151
DA 50		4.313
DA 100		57.994
DA 200		877.531

In its domain, Murϕ with its scalarset approach to symmetry, outperforms LoLA, since gathering information about symmetry is easier. However, Murϕ

has no significant reduction technique to be used with examples not featuring a dense clique-like symmetry group. This means, that if data type symmetries are applicable, they should be preferred while graph automorphism based symmetries are available for the cases where data type based symmetry does not apply.

Chapter 10

Explicit state space verification

We reconsider the classes of properties we dealt with throughout this thesis and discuss to which degree these properties are covered by explicit state space reduction techniques.

10.1 Reachability properties

Reachability is probably the class of properties with best support by explicit state space techniques. This is quite natural since reachability is the simplest class of properties concerning its temporal structure, and relates to the simplest patterns in state space search. For reachability, it is sufficient to explore the *set* of reachable state while the connectivity between different states is irrelevant. This way, we can use light weight data structures in our search algorithms. The only required capabilities of the search structure for already computed states are search and insert. Thus, fingerprint techniques for state compression can be used.

Reachability is preserved by virtually all reduction techniques discussed in this thesis. It is among the few techniques for which a simple stubborn set method exists. Reachability can therefore be verified in a setting where the search strategy can be arbitrarily chosen, including distributed search, and stubborn sets can be applied in connection with a bunch of other state space reduction techniques. Both state oriented and path oriented stubborn sets apply for reachability, the latter one, of course, only for local state predicates.

Reachability of a state predicate is supported by the symmetry method to the degree the predicate is symmetric itself. For simpler queries like the

reachability of a particular state, we can go even further and check reachability using one of the symmetry integration methods. This way, we can use a symmetry group where the checked state does not need to be symmetric.

For coverability graphs, the strongest inference rules discussed in Ch. 7 concern the temporal operators **EF** and **AG** and thus reachability as well as closely related properties.

Reachability can benefit from all of place invariants, transition invariants, and the state equation itself.

Specific instances of reachability, such as reachability of a particular state, reachability of a state enabling a transition, or reachability of deadlocks, can be verified by additional dedicated methods. On the other end of the scale, reachability is expressible in both LTL and CTL, so all known model checkers can cope with reachability properties.

In conclusion, there is excellent support for reachability verification. This is good news since a large scale of safety properties can be reduced to a reachability property through small modifications in the considered system (such as adding a monitoring process to the system).

10.2 Home properties

Home properties depend on terminal strongly connected components. We have shown that the detection of terminal components can be implemented without having a separate component stack, as required for the detection of arbitrary strongly connected components. Home properties can therefore be verified with less memory resources than using a technique that relies on arbitrary components (such as explicit state space CTL model checking). Home properties require the use of depth first search as search strategy. First, we do not have algorithms to detect terminal components using breadth first or distributed search. Second, only advances stubborn set techniques support home properties. At least, there are alternative stubborn set approaches for home properties. First, we can use dedicated home property preserving stubborn set, a class of relaxed state oriented stubborn sets. Second, we can use CTL* preserving path oriented stubborn sets. Third, we can use a two phased approach where, in the first phase, strong path oriented, and ignorance free, stubborn sets are used to find representatives of terminal components, and in the second phase any method suitable for reachability. Using the third method, other search strategies than depth first search can

be used at least in the second phase.

Symmetry reduction can be applied to home properties as long as the underlying state predicate is symmetric. Coverability analysis does not support home properties, since the combination **AGEF** is not covered by our inference rules. Interestingly, properties of the kind **EFAG**ϕ can sometimes be derived by our set of rules.

Fortunately, home property verification does not depend on restoring full state vectors out of the search structure. Thus, the place invariant based fingerprint method is applicable. Whether or not home properties can be verified in connection with transition invariant reduction must be left open, since the relation between terminal components and the incomplete information left by the transition invariant approach is not yet investigated.

All in all, dedicated explicit state space home property preserving verification method form an interesting alternative, compared to general purpose CTL model checkers (home properties cannot be expressed in LTL).

10.3 LTL model checking

In this thesis, we have not contributed to explicit state space LTL model checking. However, we would like to include LTL into our discussion, for comparison purposes. For LTL, there is a general stubborn set method preserving all LTL formulas without nextstep operator. It depends on a significant number of invisible actions, as well as on the exploration of many (in most actual implementations, all) actions at least once per cycle in the reduced state space. These two features are the main shortcoming of an otherwise successfully applied method. As symmetries preserve all CTL* $\supseteq$ LTL formulas that are insensitive to symmetry, and coverability graph preserve ACTL* $\supseteq$ LTL formulas for a certain set of atomic propositions, support for LTL verification through different explicit state space techniques is generally good. LTL verification tolerates fingerprint techniques. Whether or not it can be applied in connection with the transition invariant method, must be left open.

With our distribution algorithm, we are not yet capable of detecting cycles, so LTL model checking cannot be distributed according to our scheme.

10.4 ACTL model checking

The only known stubborn set method that supports ACTL (without the nextstep operator) is the one preserving full CTL*. This is, compared to other stubborn set methods, the weakest technique, even though it *can* lead to substantial reduction for many systems. For more complicated ACTL formulas, it becomes more and more difficult to apply symmetries. The reason is, that, at least in explicit CTL model checking, every subformula is handled by a separate search and must therefore be insensitive to symmetry in isolation.

In this thesis, we added results concerning the verification of ACTL properties by coverability graphs. This way, support for ACTL verification in the context of infinite state Petri nets improved. Even the coverability graph technique can tolerate fingerprint compression. Thus, place invariants can be beneficially applied to all reduction techniques that support ACTL.

10.5 CTL model checking

We discussed the CTL* preserving stubborn set method already in the previous section. In this thesis, we showed that state oriented stubborn sets yield alternative approaches for the two operators **AG** and **EF** in CTL that are closely related to reachability.

As long as a formula and its subformulas are insensitive to symmetry, symmetric reduction can be applied since the symmetrically reduced graph is bisimilar to the original state space thus preserving whole CTL*. With our concept of limit-satisfiability, we improved significantly on the capabilities to deduce existential CTL properties from a coverability graph. Unfortunately, nested occurrence of universally and existentially quantified formulas is not yet well supported. The explicit state space CTL model checking algorithm is capable of working in connection with the fingerprint compression based on place invariants.

The two core procedures of an explicit state space model checker—searchEU and searchAU—require the detection of strongly connected components and are thus not implementable on the basis of our distributed state space algorithm.

10.6 Liveness properties

We have seen that some frequently occurring classes of liveness properties—goal, immortality, and stabilization—can be implemented by a dedicated algorithm. Fairness constraints that are needed in order to obtain reasonable verification results can be dealt with efficiently through the algorithm in [LP85]. It is therefore desirable to find dedicated stubborn set methods for these classes of properties that are more powerful than the general LTL preserving stubborn set method that covers all three classes.

For applying symmetry, we must take care that the fairness constraints are insensitive to symmetry. This is the case in many reasonable settings.

As LTL properties, the considered classes of liveness properties are included in ACTL* thus the new capabilities of deducing universal properties from coverability graphs apply to these liveness properties as well.

Chapter 11

Discussion

In this last chapter of the thesis, we pick up selected topics that we believe deserve some final remarks.

11.1 Combined use of explicit state space methods

Throughout the chapters of the previous part of this thesis, we discussed compatibility issues between the various reduction techniques. We found that most techniques can be used in combination. Simple stubborn sets are compatible with all other reduction methods, all problems concerning symmetry or the place invariant based fingerprint techniques could be solved. Using a combined approach, we can handle systems that are another order of magnitude larger than with either reduction technique in isolation. As an example, Table 11.1 shows results for a reduced graph that has been obtained using basic stubborn sets, symmetry (with canonicalization), and fingerprint compression in combination.

For the data base example, combined application of basic stubborn sets and symmetry leads to a state space consisting of 4 states, independently of the number of involved processes. We cannot exhibit corresponding run times since the mere size of system description, and limitations in the calculation of symmetries limit the problem size we can handle. However, with a more high level formalism, and a scalarset approach to symmetry, there should be no problem with handling extremely large instances of the problem.

A combined use of coverability analysis and stubborn sets allowed us to

Table 11.1: Verification using several reduction techniques in combination: reachability of a state where two neighbours eat, and home property of the state where all but one philosopher have right fork

	PH 500	PH 1000	PH 2000
reachability (states)	1499	2999	5999
reachability (time)	25.758	137.569	835.099
home (states; phase 1 + phase 2)	1499 + 1	2999 + 1	5999 + 1
home (time)	25.100	130.914	820.308

verify boundedness or unboundedness of almost all places of a Petri net representing an unbounded version of the alternating bit protocol. In most cases, the stubborn reduced coverability graph was reasonably small if we verified boundedness of a bounded place, and returned almost immediately when we verified boundedness of an unbounded place. Interestingly, for checking boundedness of an unbounded place, stubborn reduced coverability graph generation yielded better results with depth first search than with breadth first search. The greedy heuristics inherent to the state oriented stubborn set method we used for boundedness lead depth first search approach a pumping sequence for the queried state immediately.

Playing with different combinations of available techniques, we found the following rules of thumb for combining them.

First, stubborn set methods should always be applied since the capabilities of the other techniques do not suffice to make explicit state space reduction tractable. The stubborn set method, even if not very sucessful in a particular case, decreases run time only marginally, so there is no significant advantage of switching it off. The same holds for place invariant based compression. It saves both space and time, so there is no reason to think about not applying the method. Application of coverability graph reduction should be applied if and only if there is reason to believe that the considered system has infinitely many states. In that case, there is no alternative to using coverability graphs (or any other infinite state space verification technique), while in the bounded case verification slows down too much without measurable reduction. Symmetries need to be applied with care. They require significant preprocessing and slow down state space verification substantially. In limited cases (for instance, the data base example), one can handle larger systems when not using symmetries. In most cases, however, symmetric reduction

saves enough space to compensate for the additional run time, and with canonicalization we have a symmetry integration technique that should work well in sufficiently many cases. The most involved question is whether or not to invoke transition invariant based reduction. Since it requires a parameter to be chosen carefully, we see this technique for as a last option if every other attempt to verify a system fails. The technique requires manual interaction through running several verification jobs with different parameter settings. All in all, we believe that the mutual compatibility is a big advantage for explicit state space verificatikon in contrast to symbolic verification where different methods often rely on completely incompatible data structures and algorithmic ideas.

11.2 Resource oriented versus variable oriented system description

We started this thesis with distinguishing various paradigms for describing systems implicitly. For various techniques, it turned out that that resource based formalism of Petri nets enabled reduction techniques that are not as easily available for other system description formalisms. The resource oriented view goes naturally along with a monotonous enabling condition that enables the coverability graph method, and a linear concept of invariance that enabled our approach to fingerprint compression and cycle coverage. On the other hand, we believe that variable oriented languages provide a more compact description. It should be possible to use this compact description for a more symbolic treatment of particular values of involved data types. The data type based approach to symmetry is an example where variable oriented system descriptions allow easier access to information needed in state space reduction.

As another advantage of resource based formalisms, we would like to mention the invertibility of the effect of an action. We argued that, in presence of effectively invertible actions, we can use more efficient and less space consuming data structures for representing the set of reachable states. In variable oriented approaches, it is difficult to restrict languages to invertible actions, since the constant assignment that makes an action inherently non-invertible appears to be too natural to be removed.

11.3 Explicit versus symbolic verification

As the examples in Sec. 9.9.1 suggest, explicit state space techniques can perform better than symbolic methods for some classes of examples. Of course, without needing a separate set of examples, it is immediately clear that symbolic techniques perform better for other classes. The main power horse of explicit state space verification is the family of stubborn set methods. Without a substantial amount of concurrency and asynchrony in the system, this method fails, and the other explicit state space reduction techniques can hardly compensate this unless the system exhibits, for instance, an extremely dense symmetric structure. Symbolic methods, such as BDD based model checking, do not depend on concurrency, and, though they do benefit from regularly structured systems, they can cope better with systems where symmetry is punctually disturbed.

From time to time, some kind of combination of explicit state space techniques with symbolic methods is reported. Often, their success is limited. For explicit state space reduction, the one and only criterion for success is the number of states in the reduced state space. To this goal, involved calculations such as canonicalization, or stubborn set computation can be performed for one state at a time. In symbolic verification techniques, a whole set of states is processed by one sequence of elementary operations. in order to apply explicit reduction jointly with symbolic successor set generation, it is necessary to find a sequence of operations that is valid for all simultaneously processed states. This turns out to be rather difficult, for instance for stubborn sets where calculations are highly case dependent, and those case dependencies require symbolic methods to split the sets of simultaneously processed states and continue separately [Tiu94]. Even worse, a state space consisting of less states does not necessarily have a smaller symbolic representation.

For symbolic state space verification, it is much easier to work with overapproximations than in explicit state space verification where using an overapproximation would be ridiculous, given the larger number of states. In symbolic verification, additional states can be chosen such that the symbolic representation of the state space shrinks. Other than this, there are no principal constraints for using overapproximations in symbolic state space verification since *every* ACTL* formula true of an overapproximation is true of the real state space (and it is mainly ACTL* formulas that one wants to verify). Thus, it is easier to almost freely choose an overapproximation in

order to shrink the symbolic data structure than to maintain the large list of involved conditions that are necessary to maintain a universal property in an explicit state space setting where we usually compute underapproximations.

11.4 State space versus structural verification

In structural verification, we try to deduce a property of the system from purely structural patterns in the system description. Examples are static compiler analysis, or the structure theory for Petri nets including topological considerations (such as siphons and traps) and linear algebraic techniques. Particularly in the realm of Petri net, structural verification has been studied in order to avoid expensive state space verification. Since the advent of powerful state space reduction techniques, structural techniques for Petri nets have ceased to be competitive, with the exception of linear algebraic techniques [ME96] where properties and systems are transformed in a linear optimization problem that can be solved efficiently. The topological techniques cover only a tiny portion of interesting properties. This is not a problem as such, but even on those properties they are frequently outperformed by state space techniques. The number of structural patterns (for instance, minimal siphons) to be evaluated may grow exponentially with the system size, thus requiring a large amount of run time.

Recently, it has become more and more apparent that structural verification techniques can be beneficially applied to further boost state space verification. There is some interesting technology transfer going on between static analysis and program model checking. We have shown ourselves, how the previously fully structural invariant technique can help in explicit state space verification.

11.5 Open problems

In all of the considered explicit state space techniques, we had to leave important problems open. As one of the most serious questions we would like to mention the lack of an efficient method to detect (terminal or arbitrary) strongly connected components in a distributed state space exploration. Since many powerful partial order reduction techniques, as well as

the verification of more sophisticated temporal properties rely on strongly connected components, we cannot distribute those techniques on a cluster of workstations without a component detection method. Distributed state space exploration must be further developed in other respects, too. In particlar, we need to find suitable techniques for an even distribution of load on the participating machines. We believe, that the data structure proposed in Sec. 3.3.4 is a good starting point.

For partial order reduction, we should continue the list of dedicated stubborn set methods for small classes of properties. The first candidates would be simple liveness properties such as goal, immortality, or stabilization properties, for which we have a dedicated verification scheme, but only the rather general LTL preserving stubborn set method. Additionally, it is open whether the principles of state oriented stubborn set methods can be applied to properties that are not closely related to reachability. At least, we should continue looking for a stubborn set method that does not rely on invisible actions and can thus be applied to global properties. For all advanced methods, we should try to find implementations that do not rely on strongly connected components. Perhaps, knowledge of the components can be replaced, for instance, by knowledge on transition invariants. This would be a possible way to enable distributed search in connection with advanced stubborn set methods.

Concerning the symmetry method, we need better technologies to tolerate small disturbances of the system symmetry. Initial attempts towards this goal [HITZ95] must be further investigated. It would further be interesting to combine data type and graph automorphism based techniques such that we do not lose valuable structural information about data types yet being able to benefit from the generality of the graph automorphism method. As an initial, the list of data types with support through data type symmetry should be extended more systematically, using graph automorphisms as the underlying source of inspiration.

There is nothing left to do for the place invariant approach. For transition invariants, we need to explore the relation to strongly connected components of the original state space. In general, there is a lot of ideas to explore as to employing structural verification techniques for improving state space verification.

Comparing the list of contributions in this thesis to the list of open problems, we conclude that this thesis marks just a snapshot in a dynamically evolving field.

Bibliography

[ABL96] J.R. Abrial, E. Börger, and H. Langmaak, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. LNCS 1165.* Springer Verlag, 1996.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reduction for timed systems. *Proc. CONCUR, LNCS 1466*, pages 485–500, 1998.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35 (8)*, pages 677–691, 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. *Proc. Ann. ACM Symp. Principles of Programming Language*, pages 238–252, 1977.

[CDFH90] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On well–formed colored nets and their symbolic reachability graph. *Proc. of the 11th Int. Conf. on Application and Theory of Petri Nets*, page kommt noch, 1990.

[CE82] E.M. Clarke and E.A. Emerson. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computing 2*, pages 241–266, 1982.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design 9*, pages 77–104, 1996.

[CES86a] Edmund M. Clarke, E. M. Emerson, and A. P. Sistla. Automatic verification of finite-state concurent systems using temporal logic specifications. *ACMM Transactions on Programming Languages and Systems*, 8(2):244 – 263, April 1986.

[CES86b] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8 (2)*, pages 244–263, 1986.

[CFG94] G. Chiola, G. Franceschinis, and R. Gaeta. Modelling symmetric computer architectures by swn's. *Proc. 15th Int. Conf. on Application and Theory of Petri Nets, LNCS 815*, pages 139–158, 1994.

[Chi98] G. Chiola. Manual and automatic exploitation of symmetries in spn models. *Proc. 19th International Conference on Application and Theory of Petri nets, LNCS 1420*, pages 28–43, 1998.

[DDHC92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H.Yang. Protocol verification as a hardware design aid. *Proc. IEEE Int. Conf. Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[EH86] E.A. Emerson and J.Y. Halpern. "sometimes"and "not never"revisited: on branching time versus linear time. *J. ACM 33*, pages 151–178, 1986.

[ES96] E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design 9*, pages 105–131, 1996.

[Fin90] A. Finkel. A minimal coverability graph for Petri nets. *Proc. of the 11th International Conference on Application and Theory of Petri nets*, pages 1–21, 1990.

[GKPP95] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. *3rd Israel Symp. on the Theory of Computing and Systems, IEEE 1995*, pages 130–140, 1995.

[GL83] H. Genrich and K. Lautenbach. S–invariance in Pr/T–nets. *Informatik–Fachberichte 66*, pages 98–111, 1983.

[GPMW95] R. Gerth, D. Peled, M.Y.Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. *Protocol Specification Testing and Verification*, pages 3–18, 1995.

[Gra97] B. Grahlmann. The pep tool. *Proc. Int. Conf. Computer Aided Verification, LNCS 1254*, pages 440–443, 1997.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. *6th IEEE Symp. on Logic in Computer Science, Amsterdam*, pages 406–415, 1991.

[HITZ95] S. Haddad, J.M. Iliè, M. Taghelit, and B. Zouari. Symbolic reachability graph and partial symmetries. *Proc. Int. Conf. Application and Theory of Petri nets, LNCS 935*, pages 238–257, 1995.

[HJJJ84] Huber, A. Jensen, Jepsen, and K. Jensen. Towards reachability trees for high–level Petri nets. In *Advances in Petri Nets 1984, Lecture Notes on Computer Science 188*, pages 215–233, 1984.

[Hol88] G.J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice, and Experience 18 (2)*, pages 137–161, 1988.

[Hol91] G. Holzmann. *Design an Validation of Computer Protocols.* Prentice Hall, 1991.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design 9*, pages 41–75, 1996.

[Jen81] K. Jensen. Coloured Petri nets and the invariant–method. *Theoretical Computer Science*, 14:317–336, 1981.

[Jen92] K. Jensen. *Coloured Petri Nets*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1992.

[Jun00] T. Junttila. Personal communication. 2000.

[Jun01] T. Junttila. Computational complexity of the place/transition-net symmetry reduction method. *J. Universal Computer Science 7 (4)*, pages 307–326, 2001.

[KM69] R. M. Karp and R. E. Miller. Parallel programm schemata. *Journ. Computer and System Sciences 4*, pages 147–195, Mai 1969.

[KPV97] I. Kokkarinen, D. Peled, and A. Valmari. Relaxed visibility enhances partial order reduction. *9th Int. Conf. Computer Aided Verification, Haifa, Israel, LNCS 1254*, pages 328–339, 1997.

[KV00] L.M. Krisensen and A. Valmari. Improved question-guided stubborn set methods for state properties. *Proc. 21th Int. Conf. Application and Theory of Petri nets*, pages 282–302, 2000.

[LÖ95] K. Lüttge. Zustandsgraphen von Petri-Netzen, 1995.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Toplas 16 (3)*, pages 872–923, 1994.

[LLPY97] K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. *Proc. IEEE Real-Time Systems Symp.*, pages 14–24, 1997.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *Proc. 12th Ann. ACM Symp. Principles of Programming Languages*, pages 97–107, 1985.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Springer Journal of Software Tools for Technology Transfer 1 (1,2)*, 1997.

[LS74] K. Lautenbach and H.A. Schmidt. Use of Petri nets for proving correctness of concurrent process systems. *IFIP Congress*, pages 187–191, 1974.

[MÖ1] M. Mäkelä. Optimising enabling tests and unfoldings of algebraic system nets. *Proc. 22nd Int. Conf. Application and Theory of Petri nets, LNCS 2075*, pages 283–302, 2001.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[McM02] K. McMillan. The smv home page. *http://www-cad.eecs.berkeley.edu/∼kenmcmil/smv*, 2002.

[ME96] S. Melzer and J. Esparza. Verification of system properties via integer programming. *Proc. Programming, Languages and Systems, LNCS 1058*, pages 250–264, 1996.

[Min99] M. Minea. *Partial Order Reduction for Verification of Timed Systems*. PhD thesis, Carnegie Mellon University Pittsburgh, 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Vol. 1: Specification*. Springer Verlag, 1992.

[Pel93] D. Peled. All from one, one for all: on model–checking using representitives. *5th Int. Conf. Computer Aided Verification,Elounda, Greece, LNCS 697*, pages 409–423, 1993.

[Pet73] C.A. Petri. Concepts of net theory. *Mathematical Foundations of Computer Science, Proc. Symp. and Summer School, High Tatras*, pages 137–146, 1973.

[Rei98] W. Reisig. *Elements of Distributed Algorithms*. Springer Verlag, 1998.

[RS98] S. Roch and P. Starke. INA Integrierter Netz Analysator Version 2.1. Technical Report 102, Humboldt–Universität zu Berlin, 1998.

[RV87] W. Reisig and J. Vautherin. An algebraic approach to high level petri nets. *Proc. of the 8th European Workshop on Application and Theory of Petri Nets, Zaragoza*, 1987.

[Sch96a] K. Schmidt. Ein Verfahren zur Verifikation von "immer möglich"und "möglich, dass immer"-Eigenschaften. *Workshop Algorithmen und Werkzeuge für Petrinetze*, 1996.

[Sch96b] Karsten Schmidt. *Symbolische Analysemethoden für algebraische Petri–Netze*. PhD thesis, Humboldt–Universit"at zu Berlin, vorauss.: 1996.

[Sch99a] K. Schmidt. Modelchecking with coverability graphs. *Formal Methods in System Design 15 (3)*, pages 239–254, 1999.

[Sch99b] K. Schmidt. Stubborn set for standard properties. *Proc. 20th Int. Conf. Application and Theory of Petri nets, LNCS 1639*, pages 46–65, 1999.

[Sch00a] K. Schmidt. How to calculate symmetries of Petri nets. *Acta Informatica 36,*, pages 545–590, 2000.

[Sch00b] K. Schmidt. Integrating low level symmetries into reachability analysis. *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785*, pages 315–331, 2000.

[Sch00c] K. Schmidt. Lola – a low level analyzer. *Proc. 21th Int. Conf. Application and Theory of Petri nets, LNCS 1825*, pages 465–474, 2000.

[Sch00d] K. Schmidt. Narrowing Petri net state spaces using the state equation. *Fundamenta Informaticae 47 (3,4)*, pages 325–335, 2000.

[Sch00e] K. Schmidt. Stubborn set for modelchecking the ef/ag fragment of ctl. *Fundamenta Informaticae 43 (1-4)*, pages 331–341, 2000.

[SD96] U. Stern and D.L. Dill. A new scheme for memory-efficient probabilistic verification. *Joint. Int. Conf. on Formal Description*

Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, pages 333–348, 1996.

[SD97] U. Stern and D.L. Dill. Parallelizing the murphi verifier. *Proc. Int. Conf. Computer Aided Verification, LNCS 1254*, pages 256–267, 1997.

[Sta91] P. Starke. Reachability analysis of Petri nets using symmetries. *J. Syst. Anal. Model. Simul.*, 8:294–303, 1991.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[Tiu94] M. Tiusanen. Symbolic, symmetry and stubborn set searches. In R. Valette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets in Zaragoza*, pages 511–530. Springer Verlag, 1994.

[Val88] A. Valmari. Error detection bu reduced reachability graph generation. *Proc. of the 9th European Workshop on Application and Theory of Petri Nets, Venice*, 1988.

[Val91a] A. Valmari. Stubborn sets for coloured Petri nets. In *The Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 102–121, 1991.

[Val91b] A. Valmari. Stubborn sets for reduced state space generation. *Advances of Petri Nets 1990, LNCS 483*, pages 491–511, 1991.

[Val92] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design 1*, pages 297–322, 1992.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. *5th Int. Conf. Computer Aided Verification,Elounda, Greece, LNCS 697*, pages 397–408, 1993.

[Val96] A. Valmari. Stubborn set methods for process algebras. *Proc. Workshop Partial Order Methods in Verification*, pages 213–231, 1996.

[Val98] A. Valmari. The state explosion problem. *Lectures on Petri nets I: Basic models, LNCS 1491*, pages 429–528, 1998.

[Var92] K. Varpaaniemi. On choosing a scapegoat in the stubborn set method. *Proceedings of the Workshop Concurrency, Specification and Programming Berlin 1992*, pages 163–171, 1992.

[Var98] K. Varpaaniemi. *On the stubborn set method in reduced state space generation*. PhD thesis, Helsinki University of Technology, 1998.

[VL93] B. Vergauwen and J. Levi. A linear local model checking algorithm for ctl. *Proc. CONCUR, LNCS 715*, pages 447–423, 1993.

[VW86] M.Y. Vardi and P. Wolper. An automate-theoretic approach to to automatic program verification. *Proc. IEEE Symp. Logic in Computer Science*, pages 332–344, 1986.

[WK97] R. Walter and E. Kindler. Mutex need fairness. *Information Processing Letters 62*, pages 31–39, 1997.

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. *Proc. Int. Conf. Computer Aided Verification, LNCS 697*, pages 59–70, 1993.

Index

ω-language, 70

abstract permutation, 118
action, 6
ACTL, 18
ample set method, 78
approximation, 22
arc, 12
arc weight, 12
asynchronous events, 10
average case analysis, 24

basic stubborn, 82
bisimulation, 21
boundedness, 19

canonical representative, 130
compassion, 20
composition, 9
computation tree, 16
computation tree logic, 17
constraint, 118
continuous system, 5
Convergence, 147
counterexample, 17
CTL* preserving stubborn set, 87

deterministic action, 6
deterministic system, 5
discrete system, 5
DOWN set, 88

enabled, 6, 13
event, 6

fairness, 19
finite state system, 7

goal property, 18
graph automorphism, 114
group, 113
guarded command language, 7

home property, 18
hybrid system, 5

ignored action, 84
immortality property, 19
incidence matrix, 159
initial marking, 12
initial state, 6
invertible action, 6
invisible action, 79, 83

justice, 20

key action, 82

labeled directed graph, 114
labeled transition system, 6
limit-satisfiability, 148
liveness (Petri nets), 18
liveness property, 19
location, 9
LTL, 15

LTL preserving stubborn set, 86

marking, 12
message passing, 10

node, 12
nondeterministic, 5

orbit, 113
overapproximation, 22

parallel composition, 10
Parikh vector, 159
path, 14
path quantifier, 17
permutation group, 113
persistence set method, 78
place, 9, 12
place invariant, 159
post-set, 12
pre-set, 12
progress, 20

qualitative property, 14
quantitative properties, 14

reachability property, 18
reachable, 6, 13
real-time system, 5
resource, 9
reversibility, 18

safety property, 19
scalar set, 110
scc, 35
sequential composition, 9
shared variable, 10
simple stubborn sets, 106
simulation, 21
sloppy coverability graph, 144
state, 5
state equation, 159
state explosion problem, 7
state predicate, 15
state space, 13
strict coverability graph, 143
strong fairness, 20
strong stubborn, 85
strongly connected component, 35
stubborn set, 81
stuttering insensitive, 86
symmetry integration problem, 125
synchronous events, 10
system, 5

Tarjan's algorithm, 35
temporal logic, 15
terminal state, 14
terminal strongly connected component, 57
token, 12
transition, 12
transition guard, 9
transition invariant, 160
transition relation, 8
tscc, 57

underapproximation, 22
universal property, 18
UP set, 88

visible action, 79, 83

weak fairness, 20
weak stubborn, 82
weight, 12
witness, 17
worst case analysis, 23

Lebenslauf

Titel:	doctor rerum naturalium
Staatsbürgerschaft:	deutsch
Familienstand:	ledig
Geburtstag und -ort:	3. März 1967, Berlin Friedrichshain

Schule und Studium

09/1973 - 07/1981	allgemeinbildende polytechnische Oberschule
09/1981 - 07/1985	Spezialschule mathematisch-naturwissenschaftlicher Richtung „Heinrich Hertz“, Abitur *mit Auszeichnung*
02/1981 - 08/1984	Mitglied der Mathematischen Schülergesellschaft an der Humboldt-Universität zu Berlin
09/1985 - 10/1985	Praktikum als Technischer Rechner am Rechenzentrum des Medizinischen Dienstes des Verkehrswesens
11/1985 - 08/1988	Wehrdient
09/1988 - 04/1993	Studium der Informatik an der Humboldt-Universität zu Berlin; Abschluß *mit Auszeichnung* als Diplom-Informatiker
01/1991 - 03/1992	Werkstudent bei der Bosch-Siemens Hausgeräte GmBH *Aufgabe: Simulation von Materialflußsystemen*
04/1992 - 04/1993	studentische Hilfskraft am Institut für Informatik der Humboldt-Universität zu Berlin (externes Teilprojekt YE1 im SFB 342 an der TU München)

Wissenschaftliche Tätigkeit

05/1993 - 04/1996	wissenschaftlicher Mitarbeiter bei Prof. P. Starke am Institut für Informatik der Humboldt-Universität zu Berlin
03/1996	Promotion zum *doctor rerum naturalium* mit dem Prädikat *summa cum laude*; Thema der Dissertation: „Symbolische Analysemethoden für algebraische Petri-Netze“
05/1996 - 10/1996	Stipendiat des DAAD am Digital Systems Laboratory (Prof. L. Ojala) der Helsinki University of Technology, Espoo, Finnland
11/1996 - 12/1996	wissenschaftlicher Mitarbeiter am Digital Systems Laboratory (Prof. L. Ojala) der Helsinki University of Technology, Espoo, Finnland
01/1997 - 09/1997	Stipendiat im Graduiertenkolleg „Spezifikation diskreter Prozesse und Prozeßsysteme durch operationelle Modelle und Logiken“ an der Technischen Universität Dresden (Prof. H. Vogler (Sprecher))
10/1997 - gegenw.	wissenschaftlicher Mitarbeiter bei Prof. P. Starke am Institut für Informatik der Humboldt-Universität zu Berlin
10/2000 - 09/2001	Beurlaubung zur wissenschaftlichen Tätigkeit in der *Model Checking Group* (Prof. E.M. Clarke) an der Carnegie Mellon University in Pittsburgh, PA, U.S.A.; Mitarbeit im Projekt „Model Based Integration of Embedded Software“

Forschungsgebiete

- Computergestützte Verifikation
- Petrinetze
- Echtzeitsysteme

Mitarbeit in Gremien

- Mitglied des Institutsrats Informatik der Humboldt-Universität zu Berlin 2000 - 2002
- Mitglied der Kommission Lehre und Studium des Instituts 1998 - 2002
- Mitglied der Leitung der FG 0.0.1 „Petrinetze und verwandte Systemmodelle“ der GI
- Mitarbeit im Editoral Board des Petri Net Newsletter
- Editor für die Zeitschrift Journal of Universal Computer Science (J.UCS) (Springer Verlag)
- Mitglied in den Programmkommitees der 21. und 23. „Int. Conference on Application and Theory of Petri nets“ 2000 und 2002
- Mitglied im Programmkommitee für den Workshop „Prozeßorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen“ 2002
- Mitglied im Programmkommitee für die „European Conference on Intelligent Technologies“ 2002
- Mitglied im Programmkommitee für den „5th Brasilian Workshop on Formal Methods“ 2002

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig ohne fremde Hilfe verfaßt zu haben und nur die angegebene Literatur und Hilfsmittel verwendet zu haben.

Karsten Schmidt
4. Februar 2002